Doctoral Thesis

Redesigning Transaction Processing Systems for Non-Volatile Memory

Wookhee Kim

Department of Computer Science and Engineering

Graduate School of UNIST

2019

Redesigning Transaction Processing Systems for Non-Volatile Memory

Wookhee Kim

Department of Computer Science and Engineering

Graduate School of UNIST

Redesigning Transaction Processing Systems for Non-Volatile Memory

A dissertation
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Wookhee Kim

12.12.2018
Approved by

Advisor
Sam H. Noh

Redesigning Transaction Processing Systems for Non-Volatile Memory

Wookhee Kim

This certifies that the dissertation of Wookhee Kim is approved.

12. 12. 2018

signature

Advisor: Sam H. Noh

signature

Beomseok Nam: Thesis Committee Member #1

signature

Youjip Won: Thesis Committee Member #2

signature

Woongki Baek: Thesis Committee Member #3

signature

Young-ri Choi: Thesis Committee Member #4

Abstract

Transaction Processing Systems are widely used because they make the user be able to manage their data more efficiently. However, they suffer performance bottleneck due to the redundant I/O for guaranteeing data consistency. In addition to the redundant I/O, slow storage device makes the performance more degraded. Leveraging non-volatile memory is one of the promising solutions the performance bottleneck in Transaction Processing Systems. However, since the I/O granularity of legacy storage devices and non-volatile memory is not equal, traditional Transaction Processing System cannot fully exploit the performance of persistent memory.

The goal of this dissertation is to fully exploit non-volatile memory for improving the performance of Transaction Processing Systems.

Write amplification between Transaction Processing System is pointed out as a performance bottleneck. As first approach, we redesigned Transaction Processing Systems to minimize the redundant I/O between the Transaction Processing Systems. We present LS-MVBT that integrates recovery information into the main database file to remove temporary files for recovery. The LS-MVBT also employs five optimizations to reduce the write traffics in single fsync() calls. We also exploit the persistent memory to reduce the performance bottleneck from slow storage devices. However, since the traditional recovery method is for slow storage devices, we develop byte-addressable differential logging, user-level heap manager, and transaction-aware persistence to fully exploit the persistent memory. To minimize the redundant I/O for guarantee data consistency, we present the failure-atomic slotted paging with persistent buffer cache.

Redesigning indexing structure is the second approach to exploit the non-volatile memory fully. Since the B+-tree is originally designed for block granularity, It generates excessive I/O traffics in persistent memory. To mitigate this traffic, we develop cache line friendly B+-tree which aligns its node size to cache line size. It can minimize the write traffic. Moreover, with hardware transactional memory, it can update its single node atomically without any additional redundant I/O for guaranteeing data consistency. It can also adapt Failure-Atomic Shift and Failure-Atomic In-place Rebalancing to eliminate unnecessary I/O.

Furthermore, We improved the persistent memory manager that exploit traditional memory heap structure with free-list instead of segregated lists for small memory allocations to minimize the memory allocation overhead.

Our performance evaluation shows that our improved version that consider I/O granularity of non-volatile memory can efficiently reduce the redundant I/O traffic and improve the performance by large of a margin.

Contents

List of Figures

List of Tables

I Introduction

Transaction Processing Systems(TPS) are systems that process data or operations by a logical unit, called *transaction*. A transaction is helpful in ease of implementation of program and management of data. To support this convenience, transaction should satisfy four properties called ACID (Atomicity, Consistency, Isolation and Durability) properties.

It is well known that slow storage performance is main performance bottleneck of Transaction Processing Systems from mobile to server scale. In aspects of Atomicity and Durability, Transaction Processing Systems are correlated to a storage system's performance. Transaction Processing Systems have to do logging or Copy-On-Write operations to guarantee the Atomicity of Transaction Processing Systems [1–4]. Also, successfully committed transaction should be persistently stored [1].

Emerging of Non-Volatile Memory(NVM) technologies such as NVMe, PCRAM, Intel 3D Xpoint are good candidates of replacement for storage devices in traditional transaction processing systems. Fast non-volatile memory can simply improve the performance by replacing legacy storage devices. However, the traditional systems are still optimized for page-sized block granularity of slow storage devices such as Hard Disk Drive(HDD). However, the characteristics of each non-volatile memories are different from legacy storage device. Hence simply replacing legacy storage device with non-volatile memory is hard to fully exploit the performance of newly developed non-volatile memory [2, 5].

Non-volatile memory can be divided into two types of memories, flash memory and persistent memory. Flash memory is typically block-based device. Hence it can easily replace storage devices for the page-based traditional processing systems. However, in aspects of atomic write granularity, it is different from legacy storage devices. The atomic granularity of Hard Disk Drive is sector size, but that of Flash memory is page size. Persistent memory is byte-addressable, non-volatile and as fast as DRAM. Because of the characteristics of persistent memory, it can be used as main memory and storage devices simultaneously and the use of persistent memory is actively discussed. In this dissertation, we assume that both approaches that using the non-volatile memory as a main memory(persistent memory) and as a secondary storage device (persistent memory, flash memory).

In Android devices, applications use SQLite [4] to manage their data persistently [6]. Since SQLite is an embedded database management system, when an application stores its data to a database file, the SQLite triggers its own mechanism to guarantee its data consistency. In this process, SQLite generates temporary files that include recovery information for the data. As temporary files generated and updated, file system also triggers its own mechanism to guarantee data consistency of metadata for the temporary files. These write amplification from misaligned interaction between file systems and SQLite layer called Journaling of Journal anomaly [7]. To resolve the Journaling of Journal we present LS-MVBT [8] that eliminates temporary files with Multi-version B+-tree and minimizes write traffics with optimizations. However, still, the root of

performance bottleneck is storage device. So we present NVWAL [2] that exploits the persistent memory for logging to minimize the storage overhead and file system journaling.

Since persistent memory has both advantages of DRAM and secondary block storage, we can use persistent memory as persistent main memory. As persistent memory has different I/O granularity from traditional block storage devices, the index that employed in Transaction Processing Systems also needs to be redesigned to consider the I/O granularity more carefully. CDDS B-tree [9] uses a version-based data structure that is sorted in order. However, the shift operation for sorting operation generates many cache line flush operations. To reduce the number of cache line flush operations, a bunch of papers [10, 11] proposed append-only update. Since the append-only approach does not update original data, the data can be atomically updated more easily. However, the append-only approach needs additional metadata such as bitmap, slot array [10,11], and the metadata accompanies extra cache line flushes. To minimize the cache line flush operation and store the data in sorted order, we present cache line friendly B+-tree [12] with Failure-Atomic ShifT (FAST) and Failure-Atomic In-place Rebalancing [13].

When persistent memory is used as the main memory, the main challenges we have to solve are a persistent memory leak and persistent memory fragmentation. In traditional Transaction Processing Systems, if there is memory problem such as memory leak and memory fragmentation, we can fix it by rebooting the systems. However, persistent memory stores data persistently without power supply. It means that if there is memory leak or memory fragmentation in persistent memory, the problem will be permanent. To mitigate the memory fragmentation on persistent memory, previous studies [14, 15] exploit segregated lists that divide memory chunks(256KB-2MB) into specific size classes. However, our performance studies show that these fragmentation algorithms make performance degraded. To resolve this performance degradation, we take a heap-like memory management approach.

1.1 Contributions

In this dissertation, I will show that *Byte-addressability of persistent memory can improve the performance of Transaction Processing Systems.* To support this, I developed and implemented a set of techniques and evaluated them.

The contributions of this dissertation are as follows.

- **Mitigating Write amplification between Transaction Processing Systems**

  In the Android operating systems, misaligned interaction between file system and database management system causes write amplification problem because each layer doesn't know how other layers guarantee the data consistency. This write amplification problem is called Journaling of Journal anomaly [7]. To resolve this problem, we developed LS-MVBT [8]. To our best knowledge, LS-MVBT is the first solution that solve the Journaling of Journal Anomaly. As LS-MVBT is based on multi-version B+-tree which integrates the recovery information into the main database file, we can effectively eliminate log or journal

files. To minimize the write traffics, we propose five optimizations that leverage the characteristics of Android operating systems. Moreover, we exploit persistent memory to mitigate the I/O performance bottleneck from storage devices. To minimize the write traffics, we designed three techniques, Differential Logging, User-level heap management, and Transaction-aware persistence. Our performance evaluation study shows that the combination of these techniques makes application insensitive to the latency of persistent memory [2]. We also developed failure atomic slotted paging for persistent memory that minimizes the redundant I/O for guaranteeing data consistency in a database. We show that slot header logging can guarantee the data consistency without data copy. Furthermore employing hardware transactional memory makes atomic update possible without redundant I/O [3].

- **Design a Byte-addressable Persistent Index**

  Traditional Indexing data structures are designed for block granularity based storage or main memory. The recent development of persistent memory opens up challenges for new designs that exploit both byte-addressability and durability of persistent memory. There are a bunch of works that propose persistent B+-tree [9–11, 16]. However, they still deployed page sized B+-tree nodes and append-only approach to minimize cache line flush instructions. To our best knowledge, our cache line friendly B+-tree is first B+-tree that aligns node size to I/O granularity of persistent memory. Our proposed differential encoding makes storing more data in the small size nodes possible [12]. Also, the Failure-Atomic Shift and Failure-Atomic In place Rebalancing techniques eliminate redundant I/O for guaranteeing data consistency [13].

- **Failure-atomic memory management in Persistent Memory**

  As persistent memory has better capacity than DRAM, it is expected to use persistent memory as the main memory or unified main memory and storage. When the persistent memory replaces DRAM, the memory problems in DRAM will be permanent. To prevent the permanent problem, several studies propose memory allocator for persistent memory [14, 15, 17].

  In this work, we analyzed Persistent Memory Development Kit and employed a heap-like structure instead of legacy segregated lists for small size memory allocations. In our study, the main overhead of PMDK's memory allocation comes from managing segregated lists. Our simple heap-like structure efficiently resolves the overhead of PMDK's segregated list management overhead.

1.2 Organization

The rest of this paper is organized as follows. In section II, we review the related studies to our works. Section III shows how to resolve write amplifications between Transaction Processing

Systems by exploiting Multi-version B+-tree. Furthermore, we discuss how to exploit persistent memory to reduce the redundant I/O in Section IV. In section V, we discuss the persistent index that fully exploits byte-addressability of persistent memory. Section VI presents management of persistent memory that mitigate the overhead for persistent memory allocation. Finally, we conclude this dissertation in Section VII.

II Related Works

2.1 Mitigating Write amplifiations

In Android I/O stack, SQLite [4]is a key component which allows the applications to manage their data in a persistent manner. Lee et al. [6] showed that misaligned interaction between SQLite and EXT4 file system generates an excessive I/O traffics when an Android applications stores small data.

To mitigate the excessive amount of I/O traffic, Jeong et al. [7] optimized the Android stack by employing fdatasync(), F2FS, external journaling, polling-based I/O and setting SQLite journal mode to WAL mode. However, still, WAL mode generates excessive EXT4 Journal I/O. To mitigate this, WALDIO [18] eliminates EXT4 Journal I/O with pre-allocation and minimize the logging I/O traffic with metadata embedding. With Group Synchronization, WALDIO reduces the number of fdatasync() calls. Luo et al [19], proposed qNVRAM that exploits smartphone memories as battery backed memroy to reduce the sync operation overhead.

To resolve the I/O performance bottleneck in Android I/O stack, leveraging persistent memory has been studied. SQLite/PPL [20] is a persistent per-page logging scheme for SQLite, which stores the per-page logs in PCM memory. SQLite/SSL [21] revisits logical logging to exploit the byte-addressability of persistent memory.

2.2 Byte-Addressable Persistent Index

The persistent memory technologies require redesigning of data structures that employed in transaction processing systems. Since data structures in transaction processing have an important role, they are widely studied. The most similar related works to our works are CDDS [9], NV-Tree [16], wB-tree [10], FP-Tree [11], and WORT [22]. CDDS [9] is a version-based data structure for persistent memory. CDDS allows atomic updates on persistent memory without logging. However, it generates a number of cache line flush operations to shift entries in a node. NV-tree [16] is a B-tree designed for persistent memory that reduces the number of cache line flush operations via append-only updates in leaf nodes. wB-tree [10] is a persistent b+-tree that employs append-only approaches with slot array. NV-tree and FP-tree is a b+-tree for hybrid memory systems. Both of trees take append-only approaches for leaf nodes to reduce the number of cache line flush operations and store only leaf nodes persistently. However, they need a reconstruction process for an internal node because the internal node is volatile. For concurrency control, FP-tree uses hardware transactional memory for internal nodes that reside on DRAM and use simple lock for leaf node on persistent memory. Recently Hash indexes for persistent memory have developed. Path hashing [23] is a write friendly hashing scheme that is similar to the inverted binary search tree. If a collision occurs when a data is inserted into a hash table, it will find shared space that is next level of an inverted binary search tree. This approach will reduce the space overhead, but time consumption will be increased. Level hashing

[24] is an improved hashing for persistent memory that can find data in a hash table on constant time.

2.3 Persistent Memory Management

There exist many recent works [25–31] that contributed to persistent memory(PM) management. BPFS [25] is a transactional file system for PM that adopts a short-circuit shadow paging using epoch barrier that specifies an ordering of groups of persist operations. However, the epoch barrier instruction requires a modification of memory hierarchy. NV-Heap [27] provides a way of managing persistent Memory without difficult reasoning about thread safety, atomicity, and memory access ordering. It also uses epoch barriers and mmap to allow programmers to allocate, read and write, and deallocate objects in persistent Memory. Mnemosyne [26] is a program interface that provides a set of persist primitives that consists of mfence and clflush for managing data objects in persistent memory. Heapo [29] is a memory allocator for Persistent Memory that provides programmers with simple but robust memory management interfaces. NVMalloc [28] is another persistent memory allocator that prevents memory wear-out and helps avoid erroneous memory writes and permanent system corruption. NVMDuet [30] proposed a memory scheduling policy for persistent memory, which resolves the contention between only persistent and non-persistent writes. The most similiar related works to our works is Hu et al. [31] and pAllocator [14]. Hu et al. proposed Log-structured Main Memory(LSNVMM) to minimize the fragmentation problem with log cleaning mechanism. To map the home address and log space offset, LSNVMM employs skiplist that can get rid of locks for read operation. pAllocator is a memory allocator for database management systems. pAllocator employs segregated list for small sized allocations and manage large blocks as segments with FP-Tree [11]. Makalu [32] Garbage collection approaches to persistent prevent memory leak problem.

III Multi-Version B+-tree with Lazy Split

3.1 Journaling of Journal Anomaly in Android I/O

In the Android platform, `fsync()` call is triggered by the commit of an SQLite transaction. As the journaling activity of SQLite propagates expensive metadata update operations to the file systems, SQLite spends most of its insertion (or update) time on `fsync()` function call for journal and database files [7]. The issue of resolving Journaling of Journal anomaly boils down to two technical ingredients: (i) reducing the number of `fsync()` calls in an SQLite transaction and (ii) reducing the number of dirty pages which need to be synchronized to the storage in a single `fsync()` call. Both of these two constituents eventually aim at reducing the write traffic to the block device.

In rollback journal modes (DELETE, TRUNCATE, and PERSIST) of SQLite, a single transaction consists of two phases: database journaling and the database update. SQLite calls `fsync()` at the end of each phase to make the result of each phase persistent. In EXT4 with *ordered mode* journal, `fsync()` consists of two phases: (i) writing the updated data blocks to a file and (ii) committing the updated metadata for the respective file to the journal. Most database updates in a smartphone, e.g. inserting a schedule in the calendar, inserting a phone number in the address book, or writing a note in the Facebook timeline, are less than a few hundred bytes [6]. As a result, in the first phase of `fsync()`, the number of updated file blocks rarely goes beyond a single block (4 KB). In the second phase of `fsync()`, committing a journal transaction to the filesystem journal entails four or more `write` operations, including journal descriptor, group descriptor, block bitmap, inode table, and journal commit mark, to the storage. Each of these entries corresponds to a single filesystem block.

3.2 Multi-Version B-tree (MVBT)

In an effort to reduce the number of `fsync()` calls in an SQLite transaction, we implemented version-based B-tree, *multi-version* B-tree by Becker et al. [33], which maintains update history within the B-tree itself instead of maintaining it in a separate rollback journal file (or log file). This saves SQLite one or more `fsync()` calls.

In multi-version B-tree (MVBT), each insert, delete, or update transaction increases "the most recent consistent version" in the header page of a B-tree. Each key-value pair stored in MVBT defines its own life span - $[version_{start}, version_{end})$. When a key-value pair is inserted with a new version v, the life span of the new key-value pair is set to $[v, \infty)$. When a key-value pair is deleted at version v, its life span is set to - $[v_{old}, v)$. Update transaction creates a new cell entry that has the transaction's version as its starting version $[v, \infty)$ and the old cell entry updates its valid end version to the previous consistent version number $[v_{old}, v)$. The key-value

pair whose $version_{end}$ of life span is not ∞ is called a *dead* entry. The one with infinite life span is called a *live* entry. In multi-version B-trees, the search operation is trivial. A read transaction first examines the latest consistent version number and uses it to find valid entries in B-tree nodes, i.e., if a version of a read transaction is not within the life span of a key-value pair, the respective data is ignored by the read transaction.

If a node overflows, the entries in the overflown node are distributed into two newly allocated nodes, which is referred to as "node split". An additional new node is then allocated as a parent node or an existing parent node is updated with the two newly created nodes. The life spans of the two new nodes are set to $[v, \infty)$. An overflown node becomes dead via setting the node's version range $[v_{old}, \infty)$ to $[v_{old}, v)$. In summary, a single node split creates at least four dirty nodes in version-based B-tree structures. (Please refer to [33] and [9] for more detailed discussions on the insertion and split algorithms of version-based B-tree.). In the commit phase of a transaction, SQLite writes dirty nodes in the B-tree using the `write()` system call and triggers `fsync()` to make the result of the `write()` persistent.

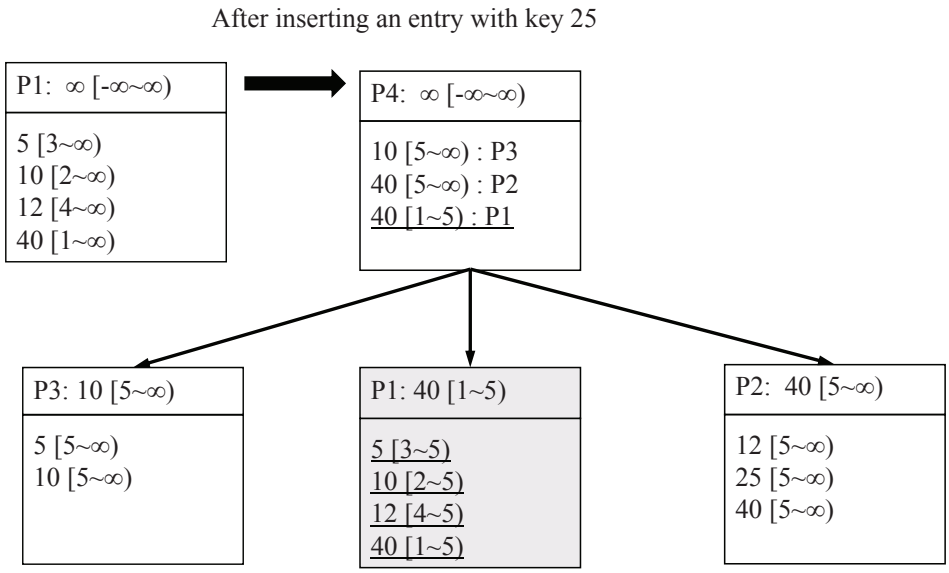
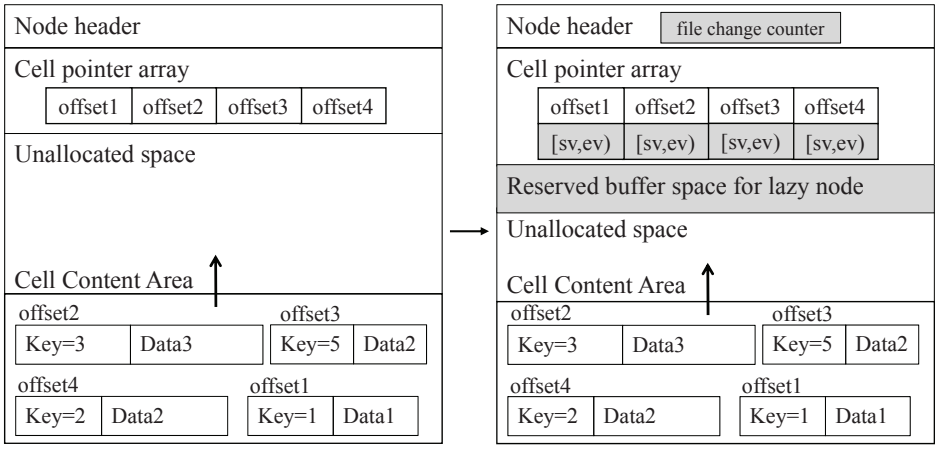


Figure 1: Multi-Version B-Tree split: After inserting an entry with key 25 into MVBT, three new nodes are created.

Figure 1 shows how an MVBT splits a node when it overflows. Suppose a B-tree node can hold at most four entries in the example. When a new entry with key 25 is inserted by a transaction whose version is 5, the node `P1` splits and a half of the live entries are copied to a new node, `P2`, and the other half of the live entries are copied to another new node, `P3`. The previous node `P1` now becomes a *dead node* and it becomes available only for the transactions whose versions are older (smaller) than 5. The two new nodes should be pointed by a parent node and the version range of the dead node should also be updated in the parent node. In the example, a new root node, `P4`, is created and the pointers to the three child nodes are stored.

The recovery in multi-version B-tree is simple and straightforward. Multi-version B-tree



(a) Layout of SQLite B-Tree Node (b) Modified Layout of SQLite B-Tree Node

Figure 2: In modified Multi-Version B-Tree node, each key-value pair is tagged with its valid starting version and ending version.

maintains the version numbers of currently outstanding transactions at the storage. In current SQLite, there can be at most one outstanding write transaction for a given B-tree [4]. In the recovery phase, the recovery module first reconstructs the multi-version B-tree in memory from the storage and determines the version number of aborted transaction. Then, it scans all the nodes and adjusts the life span of each cell entry to obliterate the effect of aborted transaction. The life span which ends at v, i.e., $[v_{old}, v)$, is revoked to $[v_{old}, \infty)$ and all cell entries which start at v are deleted.

The recent eMMC controllers generate error correction code for 4 KB or 8 KB page, hence multi-version B-tree can rely on `fsync()` to atomically move from one consistent state to the next in the unit of page size. Even if the eMMC controller promises that only single sector writes are atomic and the B-tree node size is a multiple of the sector size, multi-version B-tree guarantees correct recovery as it creates a new key-value pair with new version information instead of overwriting previous key-value pairs. A multi-version B-tree node can be considered a combination of B-tree node and journal.

3.3 Lazy Split Multi-version B-Trees (LS-MVBT)

MVBT successfully reduces the number of `fsync()` calls in an SQLite transaction as it eliminates the journaling activity of SQLite. Our next effort is dedicated to minimizing the overhead of a single `fsync()` call in MVBT. The essence of the optimization is to minimize the number of dirty nodes which are flushed to the disk as a result of a single SQLite transaction.

Multi-Version B-Tree Node in SQLite

We modified the B-tree node structure of SQLite and implemented a multi-version B-tree. Figure 2 shows the layout of an SQLite B-tree node which consists of two area: (i) *cell content*

Algorithm 1:
Lazy Split Algorithm

procedure
$LazySplit(n, parent, v)$
1: // n is an overflown B-tree node.
2: // $parent$ is the parent node.
3: // v is the version of a current transaction.
4: $newNode \leftarrow allocateNewBtreeNode()$
5: Find the median key value k to split
6: **for** $i \leftarrow 0, n.numCells - 1$ **do**
7: **if** $k < n.cell[i].key \wedge v \leq n.cell[i].endVersion$ **then**
8: $n.cell[i].endVersion \leftarrow v$
9: $newNode.insert(n.cell[i])$
10: $n.liveCells - -$
11: **end if**
12: **end for**
13: // Update the parent with the split key and version
14: $maxLiveKey \leftarrow findMaxLiveKey(n, v)$
15: $parent.update(n, maxLiveKey, \infty)$
16: $maxDeadKey \leftarrow findMaxDeadKey(n, v)$
17: $parent.insert(n, maxDeadKey, v)$
18: $maxLiveKey2 \leftarrow findMaxLiveKey(newNode, v)$
19: $parent.insert(newNode, maxLiveKey2, \infty)$
end procedure

area that holds key-value pairs and (ii) *cell pointer array* which contains the array of pointers (offsets) each of which points to the actual key-value pair. Cell pointer array is sorted in key order. In the modified B-tree node structure, each key-value pair defines its own life span - $[version_{start}, version_{end})$, illustrated as $[sv, ev)$. The augmentation with start and end version number is universal across all the version-based B-tree structures [9,33]. In our MVBT node design, we set aside a small fraction of bytes in the header of each node for *lazy split* and *metadata embedding* improvement.

Lazy Split

We develop an alternative split algorithm, *Lazy Split*, for MVBT that significantly reduces the number of dirty pages.

In MVBT, a single node split operation results in at least four dirty B-tree nodes as shown in Figure 1. The objective of maintaining a separate dead node in MVBT is to make garbage collection and recovery simple. On the other hand, creating a separate dead node yields an additional dirty page which needs to be flushed to disk. Unlike in other client/server databases, rollback operations do not occur frequently in SQLite, because SQLite allows only one process at

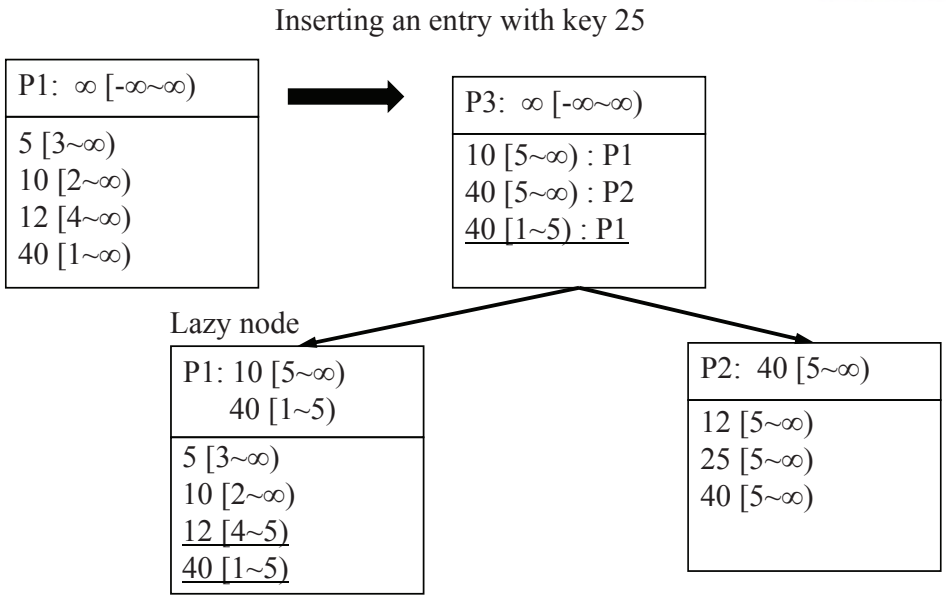


Figure 3: LS-MVBT: With the lazy split, an overflown node creates a single sibling node.

a time to have write permission to a database file [4], and rollback operations of a version-based B-tree are already very simple. Therefore, we argue that benefit of creating a separate dead node in the legacy split algorithm of MVBT hardly offsets the additional performance overhead during `fsync()` that it induces.

Algorithm 1 shows our *lazy split* algorithm that postpones marking an overflown node as dead, if possible. Instead of creating an extra dead node, lazy split algorithm combines a dead node with a live sibling node. I.e., the *lazy node* is a half dead node combined with one of the new split nodes. In the lazy split algorithm, the overflown node creates only one new sibling node. Once the median key value to split is determined, the key-value pairs whose keys are greater than the median value are copied to the new sibling node as live entries. In the overflown node, the end versions of the copied key-value pairs are changed from ∞ to the current transaction's version in order to mark them as dead entries. In the original MVBT, the key-value pairs whose keys are smaller than the median key value are copied to another new left sibling node, but lazy split algorithm does not create the left sibling node and does not change the end versions of the smaller half of the key-value pairs.

Figure 3 shows an example of lazy split. When key 25 is inserted into node `P1`, the greater half of the key-value pairs (key 12 and key 40) are moved to a new node, `P2`, and they are marked dead in `P1`. Instead of creating another new node and moving the smaller half of the key-value pairs to it, lazy split algorithm keeps them in the overflown node. The dead entries in the lazy node will be garbage collected by the next write transaction that modifies the lazy node. Note that the lazy node has two pointers pointing to it in its parent node: one for the dead entries and the other for the live entries. The same insert operation in the original MVBT will create a left sibling node, store the key 5 and key 10 in the left sibling node, and mark

the two key-value entries dead in the historic dead node as shown in Figure 1. In the example, the valid version ranges of key 5 and key 10 are partitioned in the two nodes. This redundancy does not help anything especially when we consider the short lifespan of SQLite transactions. The dead entries are not needed by any subsequent write transactions and thus can be safely garbage collected in the next modification of the lazy node because a write transaction holds an exclusive lock for the database file. The legacy split algorithm of MVBT creates four dirty nodes but lazy split decreases the number of dirty nodes by one, creating only three dirty nodes.

Reserved Buffer Space for Lazy Split

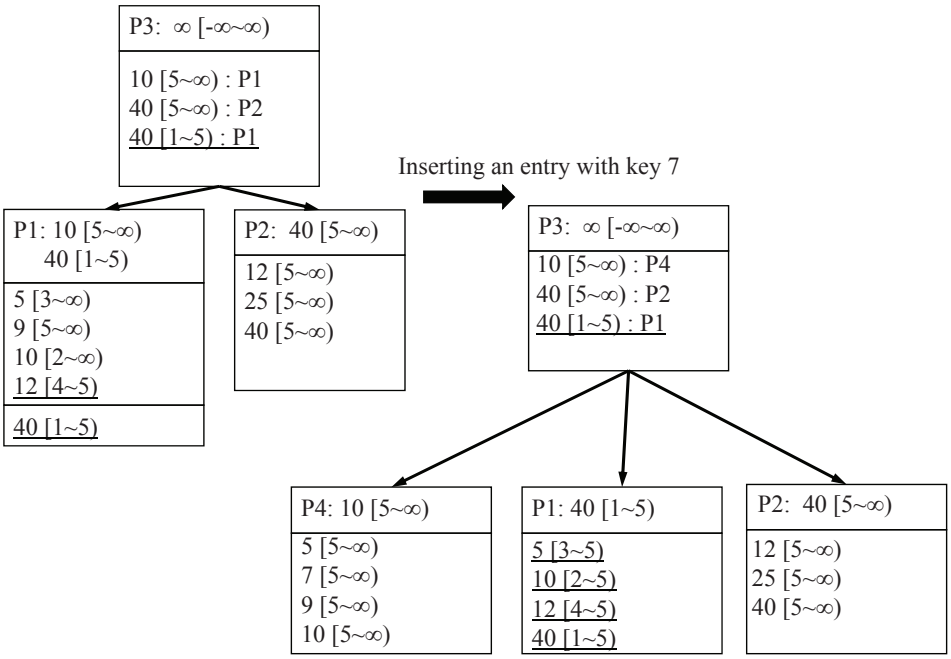


Figure 4: A new entry with key 9 is inserted into an overflown lazy node but its dead entries can not be deleted because transaction 5 is the current transaction and it may abort later. In this case, the reserved space can be used to hold the new entries and delay the node split again. But if the same transaction inserts an entry with key 7, the reserved space of the lazy node also overflows and we do not have any other option but to create a new left sibling node `P4` and move the live entries ($5[5,\infty)$, $7[5,\infty)$, $9[5,\infty)$, and $10[5,\infty)$) to `P4`.

The *lazy node* does not have any space left for additional data items to be inserted after the split. If an inserted key is greater than the median key value and is stored in a new node as in Figure 1, the lazy split succeeds. However, if a new inserted item needs to be stored in the lazy node, a new sibling node must be created as in the original MVBT split algorithm. In order to avoid splitting a lazy node, we reserve a certain amount of space in a LS-MVBT node to accommodate the inserted key in the lazy split node as shown in Figure 4.

To avoid cascade split, the size of the reserved buffer space should be sufficiently large to accommodate the newly inserted entries by a transaction. However, reserving too much space

for buffer will make node utilization low and may entail more frequent node split creating larger amount of dirty pages. The size of the reserved buffer space needs to be carefully determined considering the workload characteristics. In smartphone applications, most write transactions do not insert more than one data item. Therefore, it is unlikely that an overflown node (lazy node) is accessed multiple times by a single write transaction.

In order to evaluate the effect of the reserved buffer space size, we ran experiments varying the sizes of reserved buffer space. Large reserved buffer space is only beneficial when a single transaction inserts a large number of entries into the same B-tree node. However, a large buffer space did not significantly reduce the number of dirty nodes in our experiments, but it hurt tree node utilization especially when the B-tree node size was small. In smartphone applications, it is very common that a transaction inserts just a single data item, hence we set the size of the buffer space just large enough to hold only one key-value item throughout the presented experiments in this paper. Even if reserved buffer space for one key-value item is used, a subsequent write transaction that finds the dead entries in the lazy node will reclaim the dead entries and create empty spaces.

Rollback with Lazy Node

The rollback algorithm for the LS-MVBT is intuitive and simple. More importantly, as in the lazy split algorithm, the number of dirty nodes touched by the rollback algorithm of LS-MVBT is smaller than that of MVBT. Algorithm 2 shows the pseudo code of the LS-MVBT rollback algorithm. When a transaction aborts and rolls back, the LS-MVBT reverts its B-tree structures back to their previous states by reverting the end versions of the lazy nodes back to ∞ and deleting entries whose start versions are the aborted transaction's version. In the parent node, the lazy node has two entries: one for live entries and the other for dead entries. The parent entry of the live entries should be deleted from the parent node and the parent entry for the dead entries should be updated with its previous end version, ∞, to become active.

Figure 5 shows a rollback example. Note that node `P2` was created by a transaction whose version is 5, thus `P2` should be deleted. Since all the live entries in `P2` were copied from the lazy node `P1` by a transaction whose version is 5 and `P1` has historical entries, `P2` can be safely removed. The dead entries in `P1` should be reverted back to live entries by modifying the end versions. As the lazy node has two parent entries, the rollback process merges them and reverts back to the previous status by choosing the larger key value and by merging the valid version ranges.

3.4 Optimizing LS-MVBT for Android I/O

Lazy Garbage Collection

In multi-version B-trees, garbage collection mechanism is needed as dead entries must be garbage-collected to create empty spaces and to decrease the size of the trees. While a periodic garbage

Algorithm 2:
Rollback Algorithm

```
procedure
Rollback(n, v)
 1: // n is a B-tree node
 2: // v is the version of aborted transaction
 3: for i ← 0, n.numCells − 1 do
 4:    if n.cell[i].startVersion == v then
 5:       remove n.cell[i]
 6:       if n is an internal node then
 7:          freeNode(n.child[i], v)
 8:          continue
 9:       end if
10:       deleteEntry(n.child[i])
11:    else if n.cell[i].endVersion == v then
12:       n.cell[i].endVersion ← ∞
13:       if n is an internal node then
14:          Delete a median key entry k that was used to split the lazy node.
15:       end if
16:    end if
17:    Rollback(n.child[i], v)
18: end for
end procedure
```

collector that sweeps the entire B-tree is commonly used in version-based B-trees [9, 34], we implemented *lazy garbage collection* scheme in SQLite in order to avoid making extra B-tree nodes dirty and to reduce the overhead of `fsync()`.

When a B-tree node needs to be modified, lazy garbage collection scheme checks if the node contains any dead entries whose versions are not needed by an active transaction. If so, the dead entries can be safely deleted. The dead entries in a B-tree node will be reclaimed only when a new live entry is modified or is added to the node. Since the node will become dirty anyway by the live entry, our lazy garbage collection does not increase the number of dirty nodes at all.

Metadata Embedding

In SQLite, the first page of a database file (header page) is used to store metadata about the database such as B-tree node size, list of free pages, file change counter, etc. The file change counter in header page is used for concurrency control in SQLite.[1] When multiple processes are accessing a database file concurrently, each process can detect if other processes have changed the database file by monitoring the file change counter. However, this concurrency control design of SQLite induces significant overhead on I/O traffic since the header page must be flushed

[1]The race condition is handled by file system lock (`fcntl()`) in SQLite.

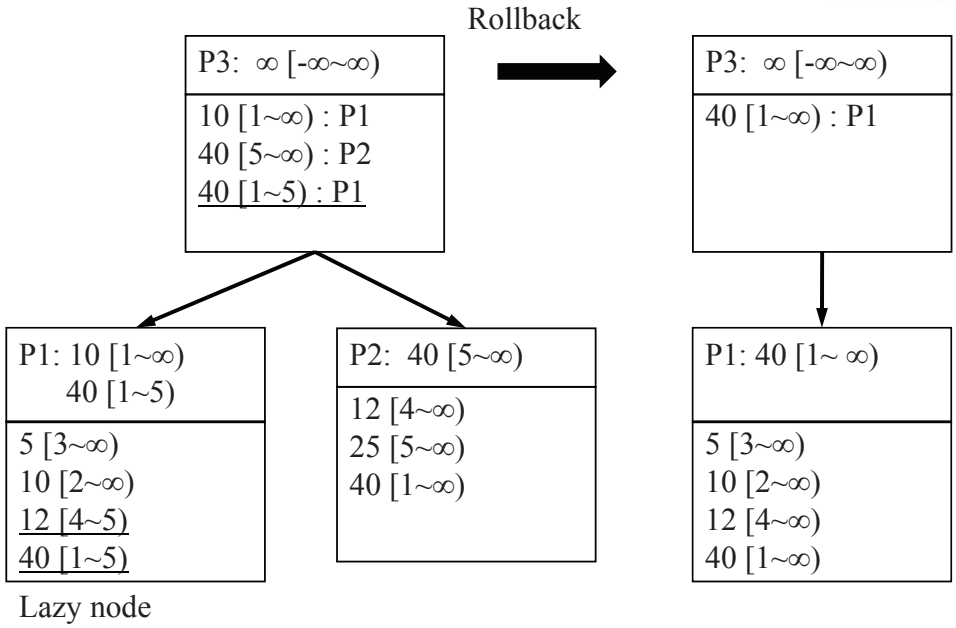


Figure 5: Rollback of transaction version 5 deletes node P2, reverts the end version of dead entries from 5 to ∞, and merges the entries in the parent node.

just to update 4 bytes of file change counter for every write transaction. This results in a large performance gap between WAL mode and the other journal modes in SQLite (DELETE, TRUNCATE, and PERSIST) since WAL mode does not use the file change counter.

In this work, we devised a method called "Metadata Embedding" to reduce the overhead of flushing database header page. In metadata embedding, we maintain the database header page at the RAM disk so that the most recent consistent and valid version ("file change counter") in the database header page is shared by transactions and the database header page is exempt from being flushed to the storage in every `fsync()` call. Since the RAM disk is volatile, the file change counter in the RAM disk can be lost. Therefore, in metadata embedding, we let the most recent file change counter be flushed along with the last modified B-tree node. When a transaction starts, it reads the database header page at the RAM disk to access the file change counter. When a write transaction modifies the database table, it increases the file change counter and flushes it to the database header page at the RAM disk and to the last modified B-tree node. Since the last modified B-tree node has to be flushed to the storage anyway, metadata embedding makes the modified file change counter persistent without extra overhead.

When a system recovers, the entire multi-version B-tree has to be scanned by a recovery process. Therefore, it is not a problem to find the largest valid consistent version number in the database and use it to rollback some changes made to the database file. If other parts of the header page are changed, we flush the header page as normal. Note that other parts of the header page are modified much less frequently than the file change counter.

Disabling Sibling Redistribution

Another optimization method used in LS-MVBT to reduce the I/O traffic is disabling redistribution of data entries between sibling nodes. If a B-tree node overflows in SQLite (and in many other server-based database engines), it redistributes its data entries to left and right sibling nodes. This is to avoid node split which requires allocation of additional nodes and changes in the tree organization. This redistribution modifies four nodes - two sibling nodes, the overflown node, and its parent node. In general, it is well known in the database community that sibling redistribution improves the node utilization, keeps the tree height short, and makes search operation faster, but we observed that it significantly hurt the write performance in the Android I/O stack.

In flash memory, time to write a page (page program latency) is 10 times longer than the time to read a page (read latency) [35] and subsequently, from SQLite's point of view, database updates, e.g., insert, update, and delete, take much longer than database search. Furthermore, search operations in smartphones are not as dominant as in client/server enterprise databases. Given these facts, we devise an approach opposite to the conventional wisdom: we disable sibling redistribution. In LS-MVBT, if a node overflows, we do not attempt to redistribute the entries in the overflown node to its siblings. Instead, LS-MVBT immediately triggers a lazy split operation.

3.5 Experiments

We implemented the lazy split multi-version B-tree in SQLite 3.7.12. In this section, we evaluate and analyze the performance of the LS-MVBT compared to other traditional journal modes and WAL mode. Our testbed is *Samsung Galaxy-S4* that runs Android OS 4.3 (Jelly Bean) on Exynos 5 Octa Core 5410 1.6GHz CPU, 2GB DDR2 memory, and 16GB eMMC flash memory formatted with EXT4 file systems.

Many latest smartphones, including Samsung Galaxy S4, adjust the CPU frequency in order to save the power consumption. We fixed the frequency to the maximum 1.6 GHz so as to reduce the standard deviation of the experiments.

The experiments section flows as follows. First, we examine the performance of SQLite transaction (`insert`) under three different SQLite modes: LS-MVBT, WAL mode, which is the default in Jelly Bean, and TRUNCATE mode, which is the default mode in Ice Cream Sandwich. Second, we take a detailed look at the block I/O behavior of SQLite transaction for LS-MVBT and WAL. Third, we observe how the versioning nature of LS-MVBT affects the search performance via examining the SQLite performance under varying mixture of `search` and `insert/delete` transactions. Fourth, we examine the recovery overhead of LS-MVBT and WAL. The final segment of the experiments section is dedicated to quantifying the performance gain of each of the optimization techniques proposed in this paper, which are lazy split, metadata embedding, and disabling sibling redistribution, in an itemized as well as in an aggregate manner.

Workload Characteristics

To accurately capture the workload characteristics of the smartphone apps, we extracted the database information from Gmail, Facebook, and Dolphin web browser apps in a testbed smartphone. Out of 136 tables in the device, the largest table contains about 4,500 records, and only 15 tables have more than 1,000 records. It is very common for smartphone apps to have such small number of records in a single database table unlike enterprise server/client databases. As most tables have less than thousands of records, we focused on evaluating the performance of LS-MVBT with rather small database tables. As for the reserved buffer space of LS-MVBT, we fix it to one cell for all the presented experiments.

Analysis of `insert` Performance

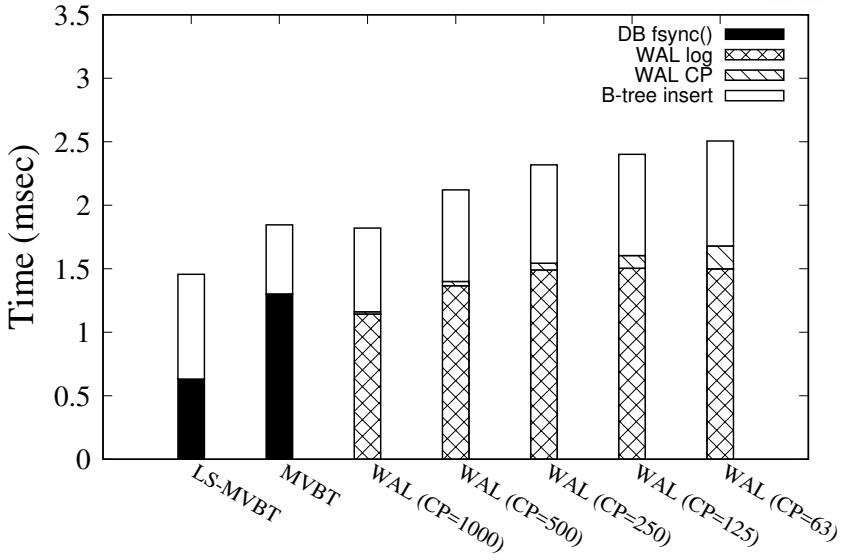
In evaluating the SQLite transaction performance, we focus on `insert` since `insert`, `update`, and `delete` generate similar amount of I/O traffic and show similar performances.

For the first set of experiments, we initialize a table with 2,000 records and submit 1,000 transactions, each of which inserts and deletes a random key value pair[2]. In WAL mode, checkpoint interval directly affects the transaction performance as well as recovery latency: with longer checkpoint interval, the transaction performance improves but the recovery latency gets longer. In SQLite, the default checkpoint interval is when 1,000 pages become dirty. The default interval can be changed by a pragma statement or a compile-time option. Checkpoint also occurs when *.db file is closed. If an app opens and closes a database file often, WAL mode will perform checkpointing operations frequently. For the comprehensiveness of the study, we vary the checkpoint intervals to 63, 125, 250, 500 and 1,000 pages. We first examine the time for a single `insert` transaction. For a fair comparison, the average insertion time in WAL mode includes the amortized average checkpointing overhead.

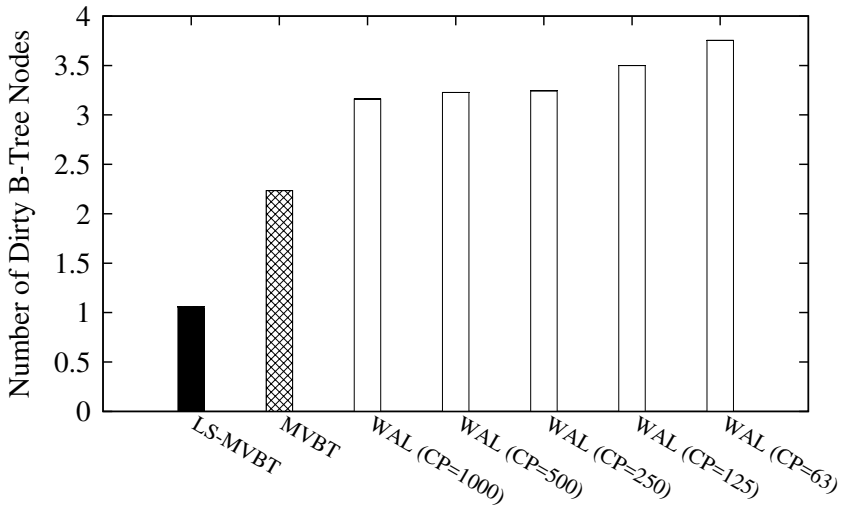
Figure 6a illustrates the result. Insertion time of MVBT and LS-MVBT consists of two elements: (i) the time to manipulate the database which is essentially an operation of updating the page content in memory, *B-tree insert*, and (ii) the time to `fsync()` the dirty pages, *DB fsync()*. Insertion time of WAL mode consists of three elements: (i) the time to manipulate the database, *B-tree insert*, (ii) the time to commit the log to storage, *WAL log*, and (iii) the time for checkpointing, *WAL CP*.

The average insertion time of LS-MVBT (1.4 ms) is up to 78% faster than that of WAL mode (2.0∼2.5 ms), but the insertion time of the original MVBT is no better than that of WAL mode. Throughout the various checkpointing intervals, LS-MVBT consistently outperforms WAL mode (even without including the checkpointing overhead). There is another important benefit of using LS-MVBT. In WAL mode, according to our measurement, the average elapsed time for each checkpoint is 7.6∼9.2 msec which is ×3 the average `insert` latency. Therefore, in WAL mode, the transactions that trigger checkpointing suffer from sudden increases in the

[2]The performance of sequential key insertion/deletion is not very different from the presented results.



(a) Insertion Time



(b) Number of Dirty B-Tree Nodes per Transaction

Figure 6: Insertion Performance of LS-MVBT, MVBT, and WAL with Varying Checkpointing Interval (Avg. of 5 runs)

latency. LS-MVBT outperforms WAL in terms of average query response time as well as in terms of the worst case bound.

We examine the number of dirty B-tree nodes per `insert` in MVBT, LS-MVBT, and WAL mode (Figure 6b). The number of dirty B-tree nodes in LS-MVBT is significantly lower than WAL mode. For an `insert`, LS-MVBT makes just one B-tree node dirty on average while WAL mode generates three or more dirty B-tree nodes. In WAL mode, not all dirty B-tree nodes are flushed to storage, but `fsync()` is called for log file commit, and the dirty nodes are flushed by the next checkpointing.

An interesting observation from Figure 6 is that the insertion performance gap between LS-MVBT and WAL is significant (40%) even when the checkpointing interval is set to 1,000 pages. When the checkpoint interval is 63 pages, the average transaction response time of WAL (2.5 msec) is 78% higher than that of LS-MVBT.

Analysis of Block I/O Behavior

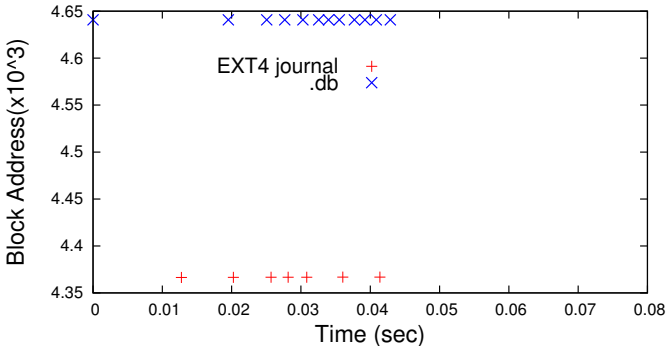
For more detailed understanding, we delve into the block I/O behaviors of SQLite transactions in LS-MVBT and WAL mode. Figure 7 shows block I/O traces of an insert operation in LS-MVBT and WAL mode. Let us first examine the detailed block I/Os in LS-MVBT. When an `fsync()` is called, the updated database file contents are written to the disk. Then, the updated metadata for the file is committed to EXT4 journal. For a single insert transaction, one 4 KB block is written to the disk for file update. Three 4 KB blocks are written to EXT4 journal, which correspond to journal descriptor header, metadata, and journal commit mark. In WAL mode, 8 KB blocks are written to the disk for log file update. Eight 4 KB blocks are written to EXT4 journal. If checkpointing occurs, there will be more accesses to a block device.

Figure 7a and 7b show the number of accesses to a block device when 10 insert transactions are submitted. Interestingly, the total number of block device accesses for 10 insert transactions in WAL mode is 84% higher than that in LS-MVBT. However, with 100 insert transactions, the number of block device accesses in WAL mode is only 46% higher than that in LS-MVBT as shown in Figure 7c and 7d. In LS-MVBT, the number of block device accesses increases linearly with the increased number of insertions whereas WAL mode accesses block devices less frequently when the size of batch insert transaction is larger.

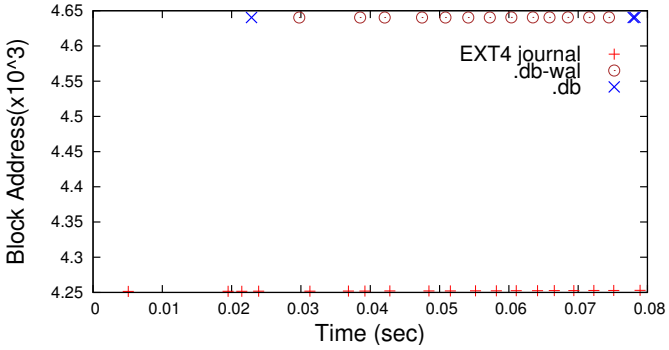
Since WAL mode writes more data than LS-MVBT per each block device access, we measure the amount of I/O traffic caused in every 10 msec. Figure 8 shows the block access I/O traffic for LS-MVBT and WAL mode. For the experiment we submit 1,000 insert transactions and measure how many blocks are accessed per every 10 milliseconds. The block access I/O traffic per 10 milliseconds for LS-MVBT fluctuates between 24 KB to 40 KB, and the EXT4 journal blocks are accessed about 24∼44 KB per 10 milliseconds. In WAL mode, the database file blocks are accessed only three times: when the database file is opened, when checkpointing occurs in 2.25 seconds, and when the database file is closed.

When the checkpointing occurs at 2.25 seconds, the I/O traffic for WAL log file increases by approximately 20 KB, from 40 KB to 60 KB, but it decreases to 40 KB when the checkpointing finishes at 2.6 seconds. In WAL mode, the number of accesses to the EXT4 journal blocks is consistently higher than any other block access types, which explains why WAL mode shows poor insertion performance. We are currently investigating what causes this high number of EXT4 journal accesses in WAL mode.

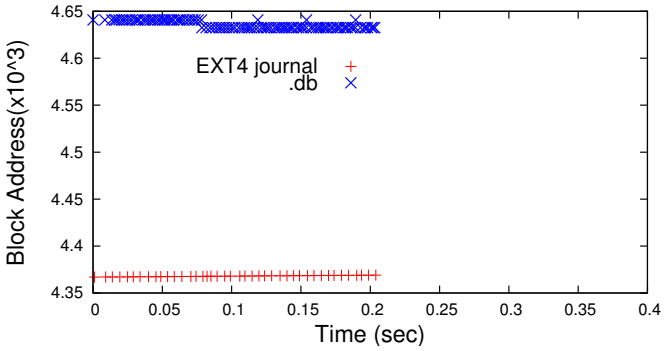
In summary, LS-MVBT accesses 9.9 MB (5 MB EXT4 journal blocks and 4.9 MB database file blocks) in just 1.8 seconds, while WAL accesses 31 MB blocks (20.7 MB EXT4 journal blocks, 9.764 MB WAL log blocks, and only 0.9 MB database file blocks) in 3 seconds.



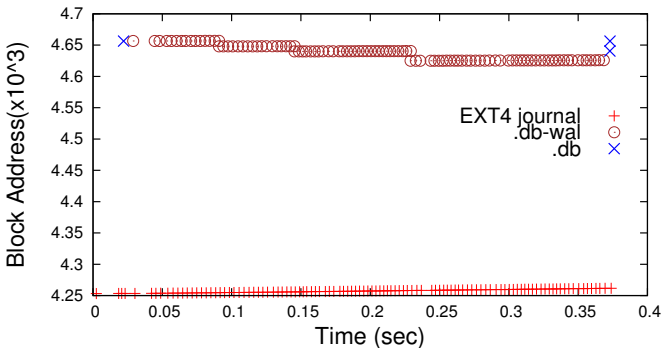
(a) Block I/O pattern of LS-MVBT (10 Transactions)



(b) Block I/O pattern of WAL (10 Transactions)



(c) Block I/O pattern of LS-MVBT (100 Transactions)



(d) Block I/O pattern of WAL (100 Transactions)

Figure 7: Block Trace of Insert SQLite Operation: LS-MVBT vs WAL

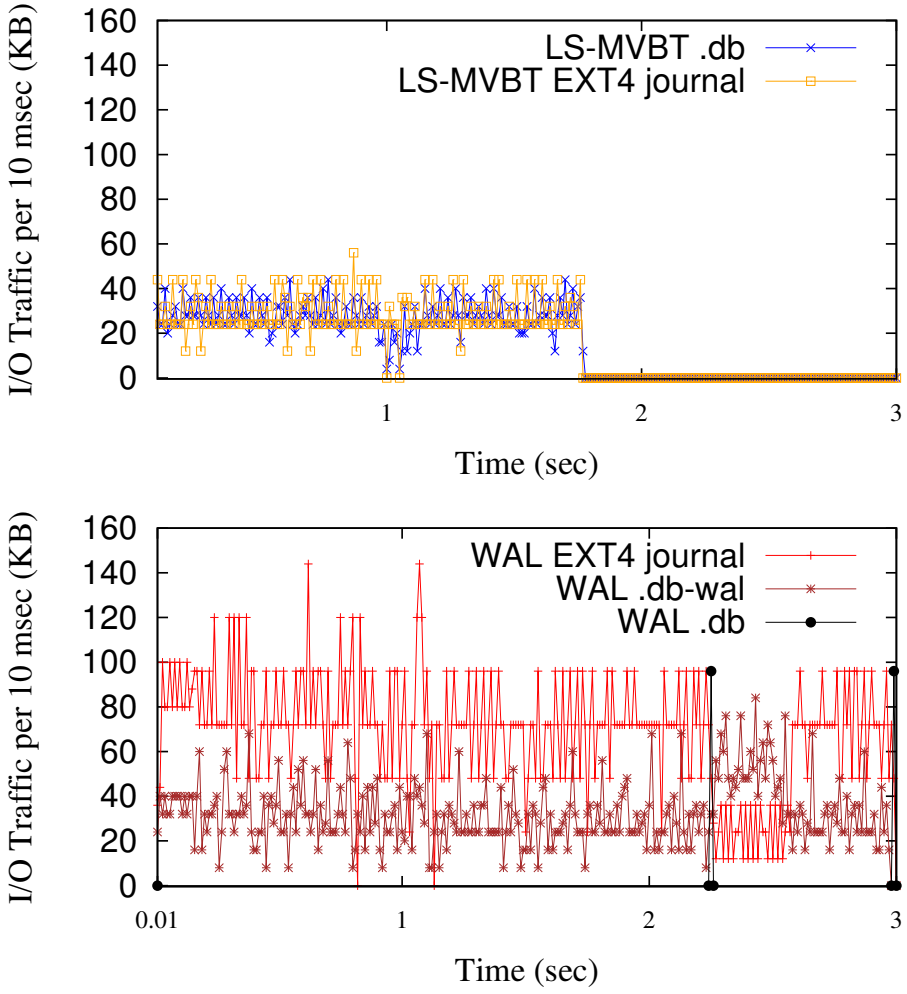


Figure 8: I/O Traffic at Block Device Driver Level (1,000 insertions)

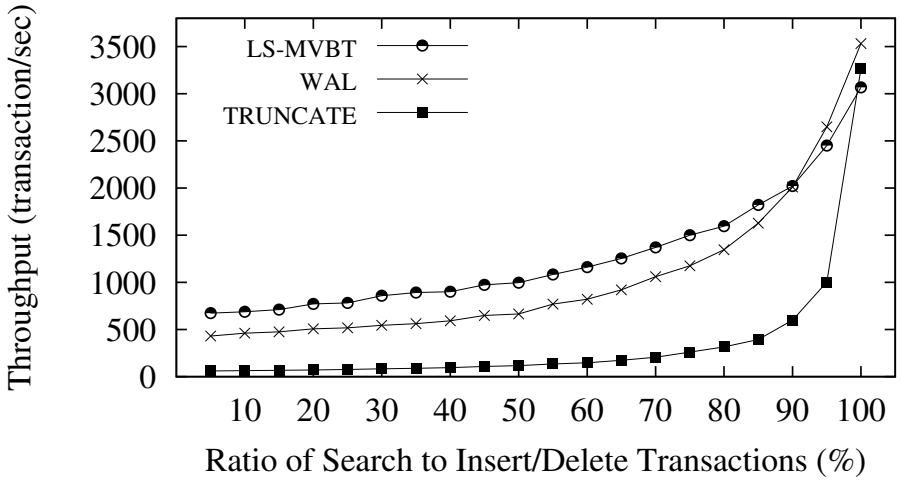


Figure 9: Mixed Workload (Search:Insert) Performance (Avg. of 5 runs)

Search Overhead

LS-MVBT makes the insert/update/delete queries faster at the cost of slow search performance. In LS-MVBT, node access has to check its children's version information in addition to the key

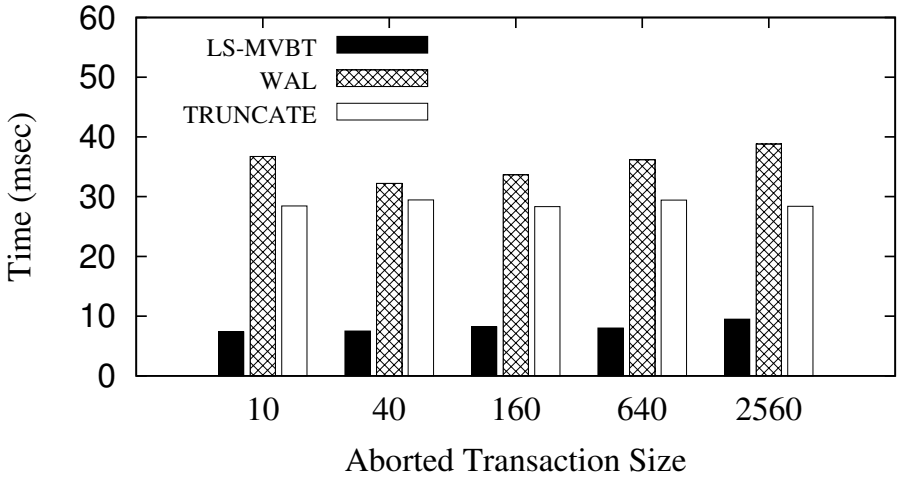


Figure 10: Recovery Time with Varying Size of Aborted Transaction

range. Moreover, LS-MVBT does not perform sibling redistribution which results in poor node utilization. Lee et al. [6] reported that write operations are dominant in smartphone applications, and the SQL traces that we extracted from our testbed device confirm this. However, the search and the write ratio can depend on individual user's smartphone usage pattern, hence we examine the effectiveness of LS-MVBT with varying the ratio of search and write transactions. We initialize a database table with 1,000 records, and submit a total of 1,000 transactions with varying ratios between the number of `insert/delete` and `search` transactions. Each `insert/delete` transaction inserts and deletes a random data from the database table, and the `search` transaction searches a random data from the table. For notational simplicity, we term `insert/delete` as `write`.

Figure 9 illustrates the result. We examine the throughput under three different SQLite implementations: LS-MVBT, WAL mode, and TRUNCATE mode. As we increase the ratio of search transactions, the overall throughput increases because a search operation is much faster than a write operation. As long as at least 7% of the transactions are writes, LS-MVBT outperforms both WAL and TRUNCATE modes. In LS-MVBT, the performance gain on write operations far outweighs the performance penalty on search operations. This is mainly due to asymmetry in latencies of writing and reading a page in NAND flash memory: writing a page may take up to 9 times longer than reading a page [35].

Recovery Overhead

Recovery latency is one of the key elements that govern the effectiveness of a crash recovery scheme. While WAL mode exhibits superior SQLite performance against the other three journal modes, i.e., DELETE, TRUNCATE, and PERSIST, it suffers from longer recovery latency. This is because in WAL mode, the log records in the WAL file need to be replayed to reconstruct the database. In this section, we examine the recovery latencies of TRUNCATE, WAL, and

LS-MVBT under varying number of outstanding (or aborted equivalently) insert statements in an aborted transaction at the time of crash: 10, 40, 160, 640, and 2560.

Figure 10 illustrates the recovery latencies of LS-MVBT, WAL, and TRUNCATE. When the aborted transaction inserts less than 10 records, WAL mode recovery takes about 4∼5 times longer than LS-MVBT. As the transaction size grows from 10 insertions to 2,560 insertions, WAL recovery mode suffers from a larger number of write I/Os and its recovery time increases by 20%. LS-MVBT recovery mode also increases by 28% but from much shorter recovery time. TRUNCATE mode recovery time slightly increases, by only 3%, but its recovery time is already 3.9 times longer than LS-MVBT when the transaction size is just 10. LS-MVBT needs to read the entire B-tree nodes for recovery but it only updates the nodes that should rollback to a consistent version.

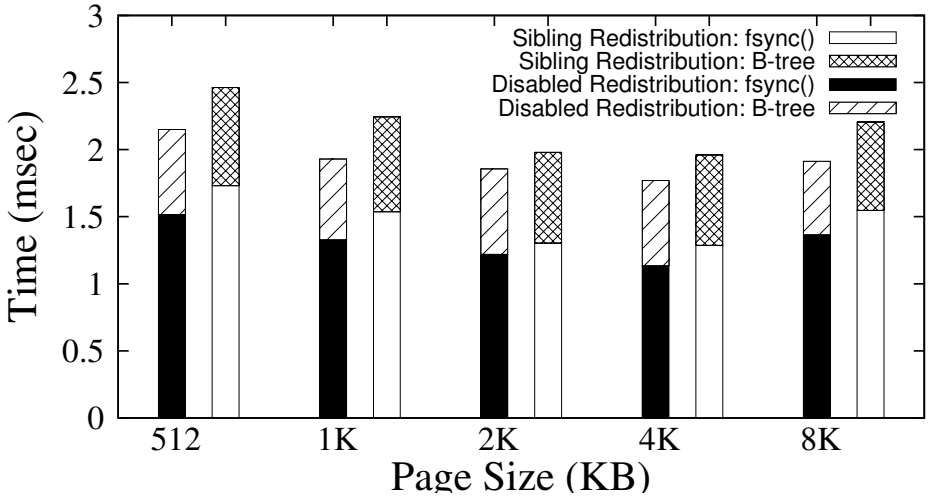
Performance Effect of Optimizations

In order to quantify the performance effect of the optimizations made on MVBT, we first examine the effect of sibling redistribution in SQLite B-tree implementation by enabling and disabling the sibling redistribution. We use the average insertion time and the average number of dirty B-tree nodes for each insertion as performance metrics in Figure 11. We insert 1,000 records of 128 bytes into an empty table, and vary the node sizes of B-tree in SQLite from 512 bytes to 8 KB.

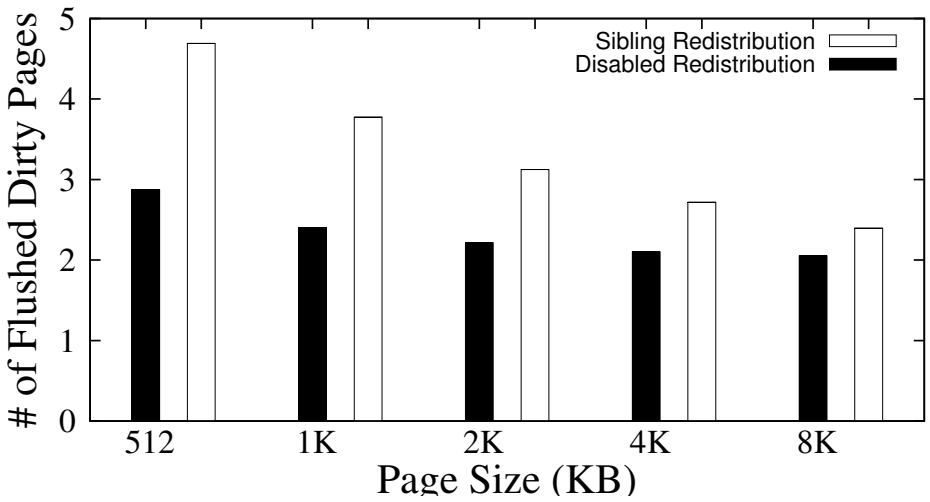
Figure 11a shows the average insertion time when sibling redistribution is *enabled* and *disabled*. When sibling redistribution is disabled, insertion time decreases as much as 20%. In the original B-tree, 70% of the insertion time is spent on `fsync()` and most of the improvement comes from the reduction in `fsync()` overhead. Figure 11b shows the average number of dirty B-tree nodes per a single `insert` transaction. With 1 KB node size, the number of dirty pages in an `insert` is reduced from 3.7 pages to 2.4 pages if sibling redistribution is disabled. Since metadata embedding can save another dirty page, with disabled sibling redistribution and metadata embedding, the average number of dirty B-tree nodes per a single insertion transaction can drop down to fewer than 2 nodes, i.e., approximately 50% of disk page flush can be saved.

With a larger node size, the number of dirty B-tree nodes decreases because node overflow occurs less often. However, we observe that the elapsed `fsync()` time grows with larger node sizes (4 KB and 8 KB) since the size of nodes that need to be flushed increases, and also the time spent in B-tree insertion code increases because more computation is required for larger tree entries. After examining the effect of B-tree node size on insert performance (Figure 11), we determine that 4 KB node size yields the best performance. In all experiments in this study, B-tree node size is set to 4 KB. [3]

[3]With 4 KB of node size, an internal tree page of SQLite can hold at most 292 key-child cells when the key is integer type while the maximum number of entries in leaf node is dependent on the record size.



(a) Insertion Time (With vs. Without Redistribution)



(b) Number of Dirty B-tree Nodes (With vs. Without Redistribution)

Figure 11: *The average elapsed time and the number of flushed dirty nodes per insertion. (Average of 1,000 insertions): Rebalancing data entries hurts write performance when a node splits.*

Putting Everything Together

It is time to put everything together and examine real world implications. In Figure 12, we compare the performance of the multi-version B-trees with different combinations of the optimizations for three different types of SQL queries. The performances are measured in terms of transaction throughput (number of transactions/sec). *MVBT* denotes the multi-version B-Tree with disabled sibling redistribution. *MVBT + Metadata Embedding* denotes the multi-version B-tree with metadata embedding optimization and disabled sibling redistribution. *MVBT + Lazy Split* is the multi-version B-tree with lazy split algorithm and disabled sibling redistribution. Finally, *LS-MVBT* denotes the multi-version B-tree with metadata embedding, lazy split algorithm, and disabled sibling redistribution. All three schemes employ lazy garbage collection and use one reserved buffer cell for lazy split. We compare these variants of multi-version B-trees

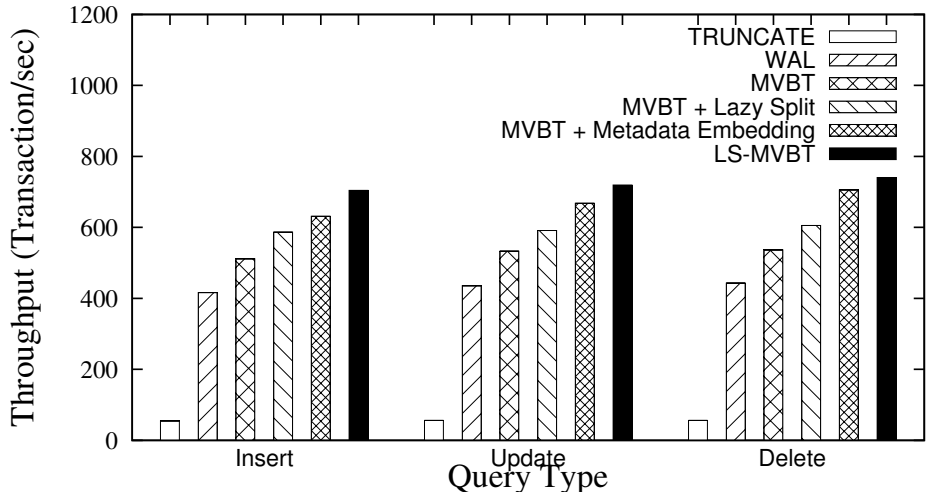


Figure 12: Performance Improvement Quantification (Avg. of 5 runs)

with TRUNCATE journal mode and WAL mode.

TRUNCATE mode yields the worst performance (60 ins/sec), which is well aligned with previously reported results [7]. Via merely changing the SQLite journal mode to WAL, we increase the query processing throughput (insertions/sec) to 416 ins/sec. Via weaving the crash recovery information into the B-tree, which eliminates the need for a separate journal (or log) file, and via disabling sibling redistribution, we achieve 20% performance gain against WAL mode. Via augmenting metadata embedding in MVBT, we achieve 50% performance gain against WAL mode.

Combining all the optimizations we propose together, (metadata embedding, lazy split, and disabling sibling redistribution), we are able to achieve 70% performance gain in an existing smartphone without any hardware assistance.

3.6 Summary

In this work, we show that lazy split multi-version Btree (LS-MVBT) can resolve the Journaling of Journal anomaly by avoiding expensive external rollback journal I/O. LS-MVBT minimizes the number of dirty pages and reduces the Android I/O traffic via lazy split, reserved buffer space, metadata embedding, disabling sibling redistribution, and lazy garbage collection schemes. The optimizations we propose exploit the unique characteristics of Android I/O subsystem: (i) write is much slower than read in the Flash based storage, (ii) dominant fraction of storage accesses are write, and (iii) there are no concurrent write accesses to database. By reducing the underlying I/O traffic of SQLite, the lazy split multi-version B-trees (LS-MVBT) consistently outperforms TRUNCATE rollback journal mode and WAL mode in terms of write transaction throughput. One future direction of this work is to improve LSMVBT in order to support multiple concurrent write transactions. With the presented versioning scheme, modifications to B-tree nodes should be made in commit order. As multicore chipsets are widely used in recent smartphones, the need

for concurrent write transactions would increase and multi-version B-tree should be improved to fully support concurrent write transactions.

IV Exploiting Persistent Memory to Minimize Redundant I/O

For the past several decades, memory technologies have evolved rapidly and now persistent memory devices such as phase-change RAM (PCRAM) and spin-transfer torque (STT) MRAM promise large capacity, high performance and low power consumption. STT-MRAM is expected to meet the requirements of various computing domains because its performance is expected to be within an order of magnitude of that of DRAM [25, 36, 37].

However, as the price of Persistent Memory(PM) is not likely to be as low as that of flash memory, it is hard to expect that persistent memory will replace flash memory in the near future. On the other hand, it can also be inferred that persistent memory will not replace volatile DRAM without major changes in the software stack: volatility of memory is inevitable in current software and hardware design because system errors will be permanent if everything is non-volatile [38]. Therefore, we design and implement write-ahead logging for persistent memory(NVWAL). Our implementation of NVWAL allows reordering of memory write operations and minimizes the overhead of the cache line flush via byte-granularity differential logging. In addition, NVWAL reduces the overhead required to manage persistent objects via user-level heap management, while guaranteeing the failure atomicity. NVWAL is not a completely novel logging data structure, but it orchestrates SQLite, the OS persistent heap manager, and persistent memory layers to effectively leverage high performance persistent Memory, as illustrated in Figure 13(b). Furthermore, we designed failure-atomic slotted paging to minimize redundant I/O. Failure-atomic slotted paging guarantees the data consistency without copying data itself. Instead, it logs small slot-header only. Also, it can update slot-header atomically by leveraging Hardware Transactional Memory.

4.1 Write-Ahead Logging on Persistent Memory

A fair number of studies have proposed to store the database transaction logs in persistent Memory [5, 39–42]. When a transaction commits, the updated pages are either synchronized to a database file in journaling modes, or appended to a log file in WAL mode. Because transactions in WAL mode do not directly update the database file but buffer the frequent small updates in the write-ahead logs, storing write-ahead logs in high performance persistent memory can replace expensive block I/O traffic with lightweight memory write instructions; such a setup can take advantage of the byte-addressability of persistent memory as well. Figure 13(a) shows how persistent Memory can be leveraged to accelerate database transactions.

4.2 Byte Granularity Differential Logging in NVWAL

In SQLite, write-ahead logging is designed to work with block device storage systems. It flushes an entire B-tree page (4 KBytes in normal SQLite configuration) to a WAL log file no matter

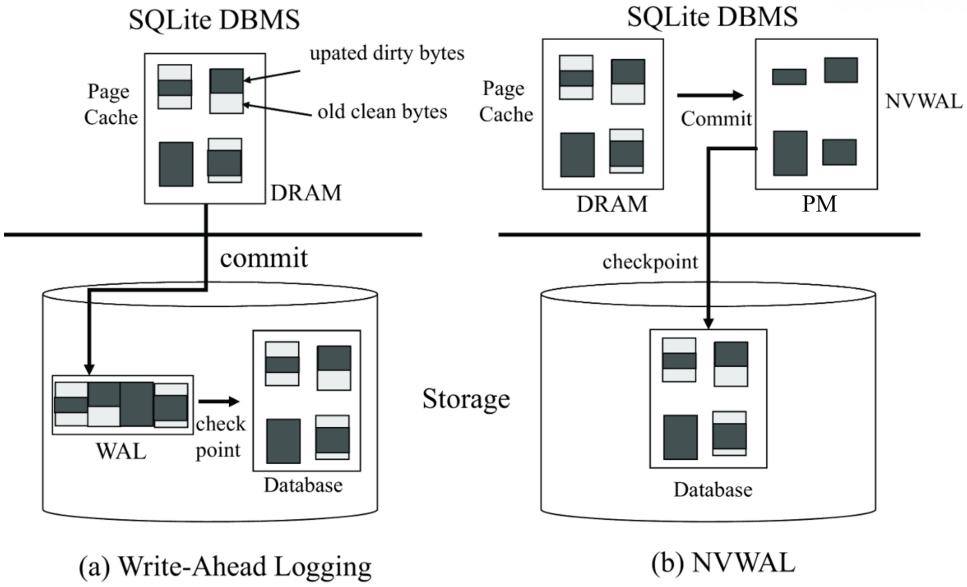


Figure 13: Write-Ahead Logging in Persistent Memory

how small a portion of the page is dirty. The size of the B-tree is aligned with the filesystem block size to avoid read-modify-write and torn-write problems in databases and file systems [43]. In most cases, a single database transaction yields changes in a small fraction of a B-tree page and leaves most of the contents of the B-tree page intact. In NVWAL, we exploit the byte-addressability of persistent memory and employ byte-granularity *differential logging* (also often referred to as *delta encoding*), which has been widely used in various systems including flash memory database systems [44–47].

As illustrated in Figure 14(b), the NVWAL structure consists of an NVWAL header and pairs of a 32 bytes WAL frame header and arbitrary-sized WAL frames (log entries) created by differential logging. The NVWAL header contains database file metadata as in the legacy SQLite WAL header and a pointer to the available space for the next WAL frame to be stored (*next_frame*). Each WAL frame header consists of a commit flag, a checkpointing id number, a database page number, an in-page offset, a frame size, and checksum bytes for the WAL frame. For each WAL frame, we identify which portion of the B-tree page is dirty and truncate the preceding and trailing clean regions so that only the dirty portions of the B-tree page are flushed to Persistent Memory, as shown in Figure 14. By minimizing the memory copy, we can avoid the unnecessary overhead of cache line flush instructions.

4.3 User-Level Persistent Memory Heap Management

There exist several well-designed proposals on how to manage persistent memory pages under a persistent namespace and how to map the pages to application processes [9,26–28,48]. BPFS [25] is a transactional file system for Persistent Memory. NV-Heap [27] and Heapo [29] are heap-based persistent object stores that allow programmers to implement in-memory data structures

Algorithm 3: `sqliteWriteWalFramesToPersistentMemory()`

```
input: WalHeader wh, PagePtr p, bool commit
while p do
    available_space = wh.getAvailableSpace() ;
    dirty_frame = compute_WAL_frame(p) ;
    if available_space < dirty_frame.size then
        /* get an persistent memory block in pending mode */
        block=nv_pre_malloc(BLOCK_SZ) ;
        ptr = wh.add_new_block(block) ;
        dmb(); /* data memory barrier */
        cache_line_flush(ptr,ptr+sizeof(void*)) ;
        dmb(); /* data memory barrier */
        persist_barrier() ;
        /* mark in-use flag of persistent memory block */
        nv_malloc_set_used_flag(block) ;
    end
    next_nv_frame = wh.next_frame ;
    /* store a dirty WAL frame in persistent memory*/
    memcpy(next_nv_frame, dirty_frame, ..) ;
    nvFramePtrList.add(next_nv_frame) ;
    p=p.next ;
end
f=nvFramePtrList.first() ;
dmb() ; /* data memory barrier */
while f do
    cache_line_flush(f,f+f.header.frame_size) ;
    f=nvFramePtrList.next() ;
end
dmb(); /* data memory barrier */
persist_barrier() ;
if commit==true then
    lastFrame = nvFramePtrList.last() ;
    lastFrame.commitMark |= COMMIT ;
    dmb() ;
    cache_line_flush(lastFrame.commitMark,...);
    dmb() ;
    persist_barrier() ;
end
```

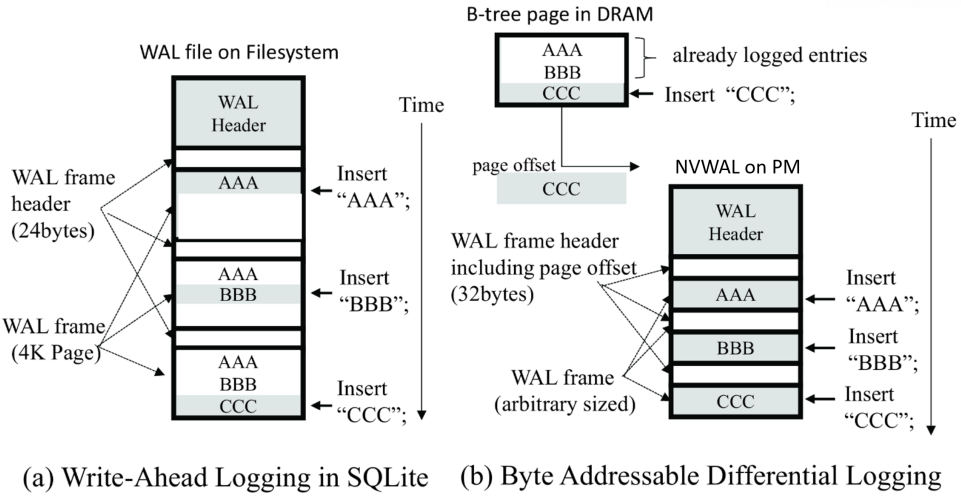


Figure 14: Byte Granularity Differential Logging

without difficult reasoning about thread safety, atomicity, or memory access ordering.

For NVWAL, we employed Heapo [4] as our persistent memory heap manager so that (i) SQLite can map persistent memory to its address spaces, (ii) a persistent memory page can be identified by a persistent namespace and its address even when system reboots, and (iii) the Persistent Memory pages can be protected by access permission as in the file system. It should be noted that Heapo does not enforce memory persistency; the applications are responsible for properly adopting a persist barrier, memory barrier, and cache line flush to guarantee the failure atomicity.

System call is expensive. It crosses the protection boundary and the parameters are copied. The system call overhead becomes even more expensive if we rely on a kernel when allocating and deallocating Persistent Memory pages. Thus, we develop a user-level heap management scheme for NVWAL. This scheme allows us to manage the log at the user level and to minimize the system call interference. Instead of relying on a kernel feature to protect an object against corruption and against the race condition, we implement a simple tri-state flag for each Persistent Memory block, i.e., *free*, *in-use*, and *pending*.

To enable user level heap management, we implemented `nv_pre_malloc()` and `nv_malloc_set_used_flag()` system calls on top of Heapo so that the system could manage the status of the allocated Persistent Memory blocks. `nv_pre_malloc()` allocates a set of Persistent Memory pages that occupy a consecutive address space, maps them to a fraction of the virtual address space of a process, and sets the status of the block to *pending*. When NVWAL persistently saves the address of a newly allocated Persistent Memory block in another Persistent Memory block as in a linked list, it calls `nv_malloc_set_used_flag()` to change its status from *pending* to *in-use*. If the system crashes while an Persistent Memory block is still in *pending* status, the SQLite NVWAL recovery process can safely deallocate the block.

[4]Heapo is available at https://github.com/ESOS-Lab/HEAPO

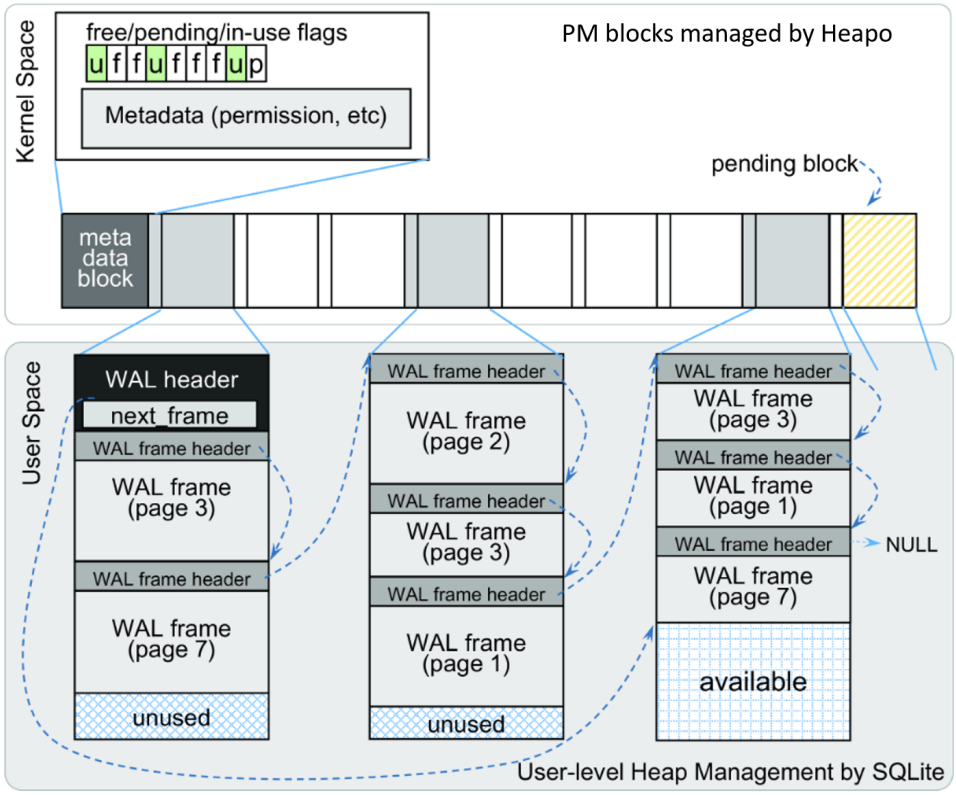


Figure 15: Persistent Memory Block Management in NVWAL

Algorithm 3 shows how NVWAL allocates and manages Persistent Memory blocks. Figure 15 illustrates an example of NVWAL block management. The NVWAL heap consists of a metadata block and a set of pages from Persistent Memory. The metadata block contains the permission and the state of each block: free, pending or in-use.

In the example shown in Figure 15, if a transaction commits a new dirty page p, SQLite searches for page p's WAL frame in the write-ahead log. If page p's WAL frame is not in the log, the entire page p and its WAL frame header are copied to the available space of the last Persistent Memory block in the linked list if they fit. If the available space in the last Persistent Memory block is not large enough to hold them, we allocate another Persistent Memory block from Heapo, add the block to the linked list of Persistent Memory blocks, set the status of the block to *in-use*, and store the dirty WAL frame and its header in the newly allocated Persistent Memory block. If page p's WAL frames are found in the log, the differences between the page and the WAL frame are computed to construct a small WAL frame, as described in section 4.2; the small differential log entry is stored as a WAL frame in the Persistent Memory log. With the pre-allocated large Persistent Memory block, NVWAL reduces the number of calls to the expensive Persistent Memory heap manager's `nvmalloc()`. In the experiments we will describe in section 4.5, each Persistent Memory block stores 4.9 WAL frames on average when we fix the size of each Persistent Memory block to 8 KBytes.

Algorithm 4: cache_line_flush() system call for ARM v7

```
input: u32 start, u32 end
u32 lineLen = getCachelineLen();
/* Align start to cache line boundary*/
start = start & ∼(lineLen - 1);
u32 mva = start;
while mva < end do
    /* clean data cache line to PoC by MVA */
    dccmvac(mva) ;
    mva = mva + lineLen ;
end
```

4.4 Transaction-Aware Memory Persistency Guarantee

When persistent memory is used to replace block device storage, the atomicity and the durability of the database transactions must be ensured in the memory operations. NVWAL requires that we explicitly flush the appropriate cache lines to persistent memory to enforce the ordering constraints. The WAL recovery algorithm is based on the assumption that log entries are stored in the order of database transactions. Unfortunately, memory write operations can be arbitrarily reordered in today's processor designs. Without the hardware support `sfence` or `mfence` alone cannot guarantee the propagating of memory writes to persistent memory because writes can be cached in L1 or L2 caches.

In order to guarantee that memory write operations flush data all the way down to persistent memory, cache line flush operations such as `clflush` should accompany `persist barrier` instructions as shown in Figure 16(b). `Persist barrier` ensures that the cache lines queued in the memory subsystem are persisted in persistent memory. Otherwise, a commit mark can be written to persistent memory before log entries are stored.

In X86 processors, `clflush` instruction invalidates and flushes a cache line from the memory cache hierarchy. In the ARM v7 architecture, there exist two cache flush instructions: `dccimvac` and `dccmvac`. The former invalidates the flushed cachelines while the latter does not. Android 4.4 has not implemented the `clflush()` system call. For NVWAL, we implemented a `cache_line_flush()` system call using a `dccmvac` instruction, as shown in Algorithm 4. `cache_line_flush()` calls `dccmvac` instead of `dccimvac` because write-ahead logging in SQLite does not, indeed, need cache invalidation due to its append-only nature. We implemented `cache_line_flush()` as a system call despite the overhead of the kernel mode switch because the `dccmvac` instruction needs to access register 15 in privileged mode.

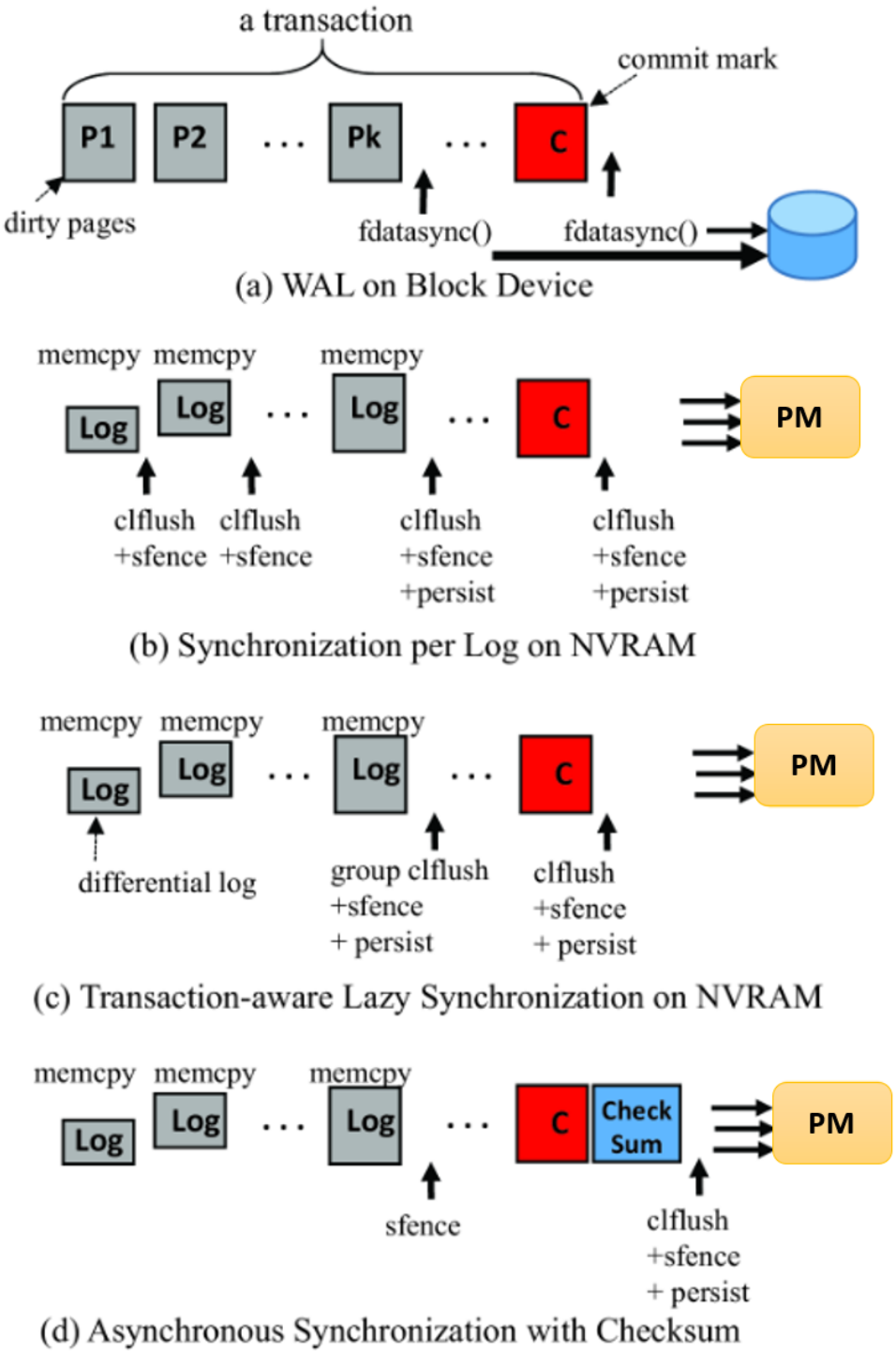


Figure 16: Transaction-Aware Persistency Guarantee

Transaction-Aware Memory Persistency Guarantee

We exploit the ordering and persistency constraints in write-ahead logging and develop a transaction-aware memory persistency guarantee scheme. The idea is simple and straightforward. A log-commit operation consists of two phases: (i) logging: writing a sequence of logs to persistent memory and (ii) commit: putting the commit mark to persistent memory to validate the logs. We enforce the ordering and persistency guarantee requirement only between the two phases by calling the expensive `clflush` and `sfence`, and `persist barrier`. The reason behind this is that the ordering of writing WAL frames does not necessarily have to be preserved as long as a transaction's commit mark is written after all the WAL frames are written to persistent memory. It should be noted that (i) SQLite is a serverless DBMS that does not allow multiple write transactions to run concurrently and (ii) a write transaction requires an exclusive lock on the entire database file.

Under a transaction-aware memory persistency guarantee, so that the processors can better utilize caches and memory banks, NVWAL allows processors to flush the WAL frames to persistent memory in any order. That is, NVWAL separates the phase in which the logs are being copied to persistent memory (`memcpy()`) and the phase in which the respective logs are synchronized to persistent memory (`cache_line_flush(p)`). Figure 16(c) schematically illustrates the behavior of the transaction-aware memory persistency guarantee. The details of the implementation can be found in Algorithm 3.

There are a few implementation specific issues that deserve further elaboration. We implemented NVWAL in ARM based platform. The cache line flush instruction in an ARM, `dccmvac`, is non-blocking. In order to ensure that the memory operations that mark the commit flag (line 30–34) are not reordered with the following `cache_line_flush()`, the cache line flush instruction should be preceded by the data memory barrier, `dmb`. The `dmb` instruction completes only when any previous explicit memory access instructions are all completed. The `dmb()` in line 22 ensures that all WAL frames are written at least to the L1 or L2 cache. In the proposed *lazy synchronization*, a batch of non-blocking `dccmvac` instructions are called in line 23–26 in order to flush the cache lines. The `dmb()` in line 27 is also needed in order to block until the preceding `cache_line_flush()` flushes all the WAL frames to persistent memory. `dmb()` in line 32 also ensures that the memory operation that marks the commit flag is not reordered with `cache_line_flush()`.

The commit mark resides at the WAL frame header and is one bit long. As in a prior work [25], we assume that persistent memory devices guarantee atomic writes for 8 bytes. This means that even if a random power failure occurs, capacitors in DIMM guarantee no corruption of 8 bytes. Because the commit mark is just a bit flag, the commit mark can be safely flushed to persistent memory with 8 bytes padding. If atomic writes can be done in the unit of a cache line, as in [28], cache line padding is necessary to prevent `cache_line_flush()` from flushing an unintended memory region.

Asynchronous Commit

We can further allow processors to utilize caches and memory banks by compromising the consistency using checksum bytes. Under this mechanism, we do not explicitly enforce the barrier between the logging phase and the commit phase. Instead, we compute the checksum of the logs that must precede the commit mark and store the checksum along with the commit mark. Figure 16(d) illustrates NVWAL with asynchronous log-commit; NVWAL asynchronously writes log entries, a commit mark, and checksum bytes without calling the cache line flush instructions.

Suppose a system crashes after it flushes a commit mark and checksum bytes but before all log entries are written to persistent memory. In the recovery phase, the checksum bytes will be found to be inconsistent with the written log entries and this will invalidate the commit mark of the transaction. Thus, the database recovery process can consider that the transaction has been aborted. However there is a chance that the written checksum bytes accidentally match the unwritten log entries. Hence, although the chance is very low, a system crash may corrupt a database file. We consider this asynchronous synchronization NVWAL as a performance comparison target because it minimizes the overhead of enforcing memory constraints.

Checkpointing and Crash Recovery

In WAL, the committed dirty pages in persistent memory are written back to the original database file in block device storage via periodic checkpointing. In NVWAL, we reconstruct the dirty pages by combining the log entries in the NVWAL log and the respective original database pages. When all the dirty pages are flushed to the database file via `fsync()`, the persistent memory blocks for the write-ahead logs can be safely truncated from the end of the list to the beginning. For each persistent memory block, we first set the tri-state flag of the persistent memory block to *free* and then call Heapo's `nvfree()` system call.

In SQLite WAL mode, the log file contains a complete history of dirty pages committed by transactions. When a system crashes, SQLite can recover the transactions by replaying the logs. The recovery algorithm of NVWAL is not very different from SQLite's stock recovery algorithm except that the dirty pages are reconstructed from byte-addressable WAL frames. However, complications can arise if the system crashes while updating the metadata of the persistent memory blocks.

In order to proof of correctness for recovery algorithm of NVWAL, we resort to discuss the persistent memory recovery algorithm under various failure cases that can occur while a transaction is executing `sqliteWriteWalFramesToPersistentMemory()`.

- If a system fails while allocating an persistent memory block (line 6 of Algorithm 3), the allocated persistent memory block is marked as *pending*, but SQLite may not have written its reference in another persistent memory block. When the system recovers, the heap manager can reclaim any pending persistent memory blocks to prevent a memory leak.

- If a system crashes after an persistent memory reference was stored in the WAL header but before the block was marked as in-use, the SQLite recovery process will find that the block was freed by the persistent memory heap manager's recovery process and the block's reference can be safely deleted.

- If system crashes while copying a dirty WAL frame to persistent memory using `memcpy()` (line 17 of Algorithm 3), SQLite can easily recover from the failure because the frame's transaction has not written a commit mark, thus the transaction is considered to have been aborted. In database transactions, writing a commit mark is the last operation of the transaction commit process.

- Recovery from checkpointing process' failure is also trivial and not different from legacy checkpointing recovery. Because the write-ahead logs will not be deleted before all the dirty pages are persistently stored in the database file, the SQLite recovery process can simply replay the checkpointing process to recover from the failure.

NVWAL and Persistency Model

We briefly discuss NVWAL implementation for strict and relaxed persistency models.

Memory persistency, proposed by Pelley et al [37], is a framework that provides an interface for enforcing the ordering constraints on persistent memory writes. The ordering constraints are referred to as "persists" to distinguish them from regular writes to the volatile memory [37]. Similar to memory consistency, memory persistency is largely classified into two classes – *strict persistency* and *relaxed persistency*.

Strict persistency integrates memory persistency into the memory consistency [37]. Under strict persistency, persist order must match the volatile memory order specified by the memory consistency model. Strict persistency is a simple and intuitive model in the sense that it provides a unified framework to reason about possible volatile memory and persist orders. In addition, it requires no additional persist barriers because the existing memory barriers can be used to specify persist ordering constraints. Strict persistency, however, may significantly limit persist performance because it enforces strict ordering constraints between persist operations.

In contrast, relaxed persistency decouples the memory consistency and persistency models [37]. Under relaxed persistency, persist order may deviate from the volatile memory order specified by the memory consistency model. Relaxed persistency requires persist barriers to enforce the order of persist operations. A representative relaxed persistency model is *epoch persistency* [25, 37]. In epoch persistency, persist barriers are used to divide persist operations into different persist epochs. Persist barriers ensure that all the persists before the barrier occur strictly before any persist after the barrier. Persists in the same epoch, however, are allowed to concurrently execute, potentially improving persist performance. A major disadvantage of relaxed persistency is the increased programming complexity; programmers must correctly annotate their code using the two types of barriers (i.e., memory and persist barriers).

With memory persistency, it is the responsibility of the underlying hardware to enforce the ordering constraints of persistent memory writes specified in the (annotated) code. Therefore, no extra code is required to explicitly flush appropriate cache lines to persistent memory, easing the programmer's burden.

Strict persistency significantly simplifies the NVWAL implementation shown in Algorithm 1 because all the cache-flush operations and persist barriers can be safely removed. However, we conjecture that strict persistency may degrade the performance of NVWAL because it enforces strict (but unnecessary) ordering constraints between persists when writing the log entries to persistent memory.

Relaxed persistency simplifies the NVWAL implementation shown in Algorithm 1 because all the cache-flush operations can be safely removed. We expect that relaxed persistency, because it can dynamically reorder persist operations when copying the WAL frames from DRAM to persistent memory, will induce a level of performance for NVWAL higher than that possible when using strict persistency. Due to the unavailability of real hardware that can implement strict and relaxed persistency, we leave a performance evaluation of NVWAL under various memory persistency models to our future work.

4.5 Experiments

We implemented NVWAL in SQLite 3.7.11 and integrated it with an persistent memory heap manager - *Heapo* [29]. We first measure the performance of NVWAL using a *Tuna* persistent memory emulation board [49, 50].

Tuna is an persistent memory emulation board with Xilinx Zynq XC7Z020 ARM-FPGA SoC. The Tuna board consists of an ARM Cortex-A9 processor with L2 caches and FPGA programmable logic that controls the read/write latency of one of the two DRAMs in order to emulate persistent memory. The emulated persistent memory has a separate power switch for non-volatility emulation. The frequency of the emulated persistent memory is 400MHz (DDR3-800) while that of the volatile DRAM is 533MHz (DDR3-1066); the data width of the emulated persistent memory is 64 bits while that of DRAM is 32 bits. The size of the cache line is 32 bytes. The write latency of the emulated persistent memory can be adjusted between 400 nsec and 2000 nsec.

Overhead of Ordering Constraints

First, we quantify the overhead of enforcing the ordering constraints in SQLite write-ahead logging. In the configuration denoted as E (eager synchronization), we make SQLite write-ahead logging call `cache_line_flush()` system call and memory barrier immediately after each `memcpy()` function call per log entry as illustrated in Figure 16(b). In the other configuration, denoted as L (lazy synchronization), SQLite calls a batch of `cache_line_flush()` system calls

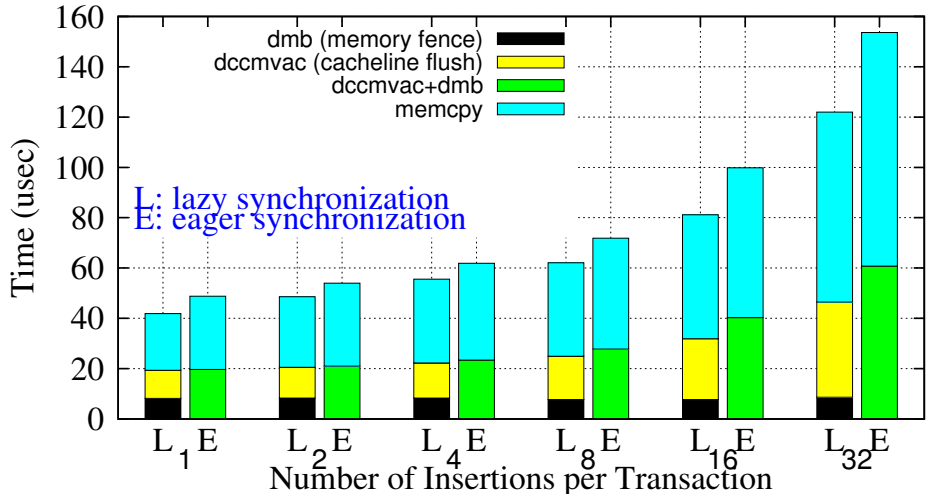


Figure 17: Quantifying the benefit of allowing processors to reorder memory writes

# of insertion per txn	1	2	4	8	16	32
# of cache line flushes	139.49	153.32	181.224	236.52	349.168	574.464

Table 1: Average number of cache line flushes per transaction for the experiments shown in Figure 17

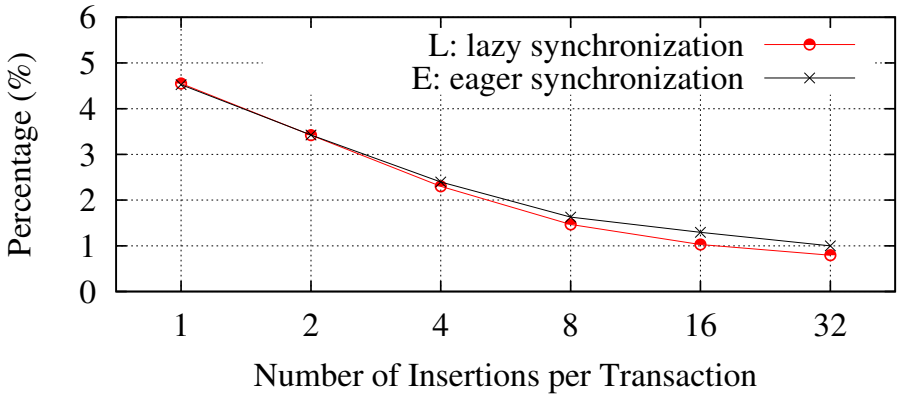


Figure 18: Proportion of ordering constraint overhead to query execution time

for all the dirty log entries in a transaction right before its commit mark is stored, as illustrated in Figure 16(c).

We run the experiments on a Tuna persistent memory emulation board and set the write latency of the persistent memory to 500 nsec as in [37]. Figure 17 shows the benefits of lazy synchronization. As we insert more records into the database table in a single transaction, more dirty bytes are written to persistent memory and the time spent for memory writes increases. Table 1 shows how many cache lines are flushed per transaction (the number of called dccmvac instructions) with varying the number of insertions per transaction. When each transaction inserts a single data record, the time spent on `dccmvac`, `dmb`, and kernel mode switching is only

# of operation per txn	1	2	4	8	16	32
Insert	4431.8	4874.2	5767.1	7536.6	11141.1	18350.0
Insert (Diff)	726.8	863.0	1139.7	1684.0	2776.0	4984.0
Delete	4890.6	5685.2	7274.4	10452.9	16842.7	22282.2
Delete (Diff)	1555.2	2236.8	3576.9	5863.0	9412.6	11452.9
Update	4096	4210.6	4440.0	4882.4	5832.7	7733.2
Update (Diff)	647.7	830.3	1186.5	1857.9	3191.5	5529.6

Table 2: Average number of bytes written to Persistent Memory

19.3 usec per transaction. Considering that the query execution time is 424 usec, this overhead is just 4.6% of the query execution time as shown in Figure 18. However, when a transaction inserts 32 data records, the query execution time is 5828 usec while the overhead of the write ordering constraints is 46.5 usec (only 0.8 % of the execution time). SQLite throughput is governed more by the computation performance than by the I/O performance, especially when a WAL file is located in fast persistent memory. Hence, improving the I/O performance by reordering the memory writes does not significantly affect the ratio of the ordering constraint overhead to the query execution time, as shown in Figure 18.

In eager synchronization, denoted as *E*, we do not allow reordering of memory writes. But, in lazy synchronization denoted as *L*, we call `memcpy()` for all dirty portions of the WAL frames in a transaction before NVWAL explicitly calls `dccmvac`. Therefore the dirty bytes in the L1 or L2 cache could have been already evicted. I.e., in lazy synchronization, the overhead of `dccmvac` is masked by the overhead of `memcpy()`. As in epoch persistency, the lazy synchronization decouples the volatile memory order from the persist order, which fits quite well with the database transaction concept.

The amounts of time spent on `memcpy()` in both schemes are similar in the experiments, as shown in Figure 17. However, in *E*, the `dccmvac` and `dmb` together perform up to *23% slower* than does `dccmvac` in *L* because *E* does not allow the reordering of memory writes. By separating `memcpy()` and `dccmvac`, write-ahead logging can eliminate about 2∼23% of the total overhead of enforcing persistency (19.3 ∼ 46.4 usec vs. 19.7 ∼ 60.7 usec). The overhead of `dmb` in *L* is negligible because `dmb` is called at most only four times in our implementation.

Differential Logging and I/O

Table 2 shows the I/O volume written to persistent memory with byte granularity differential logging and with legacy block granularity logging. For the insert, update, and delete operations, byte-granularity differential logging eliminates 73∼84%, 29∼85%, and 49∼69% of the unnecessary I/Os to persistent memory, respectively. In SQLite, each B-tree page appends a newly inserted set of data to the end of the used region. Thus, the size of a WAL log entry generated by byte-granularity differential logging is often small. But, for the update and delete operations,

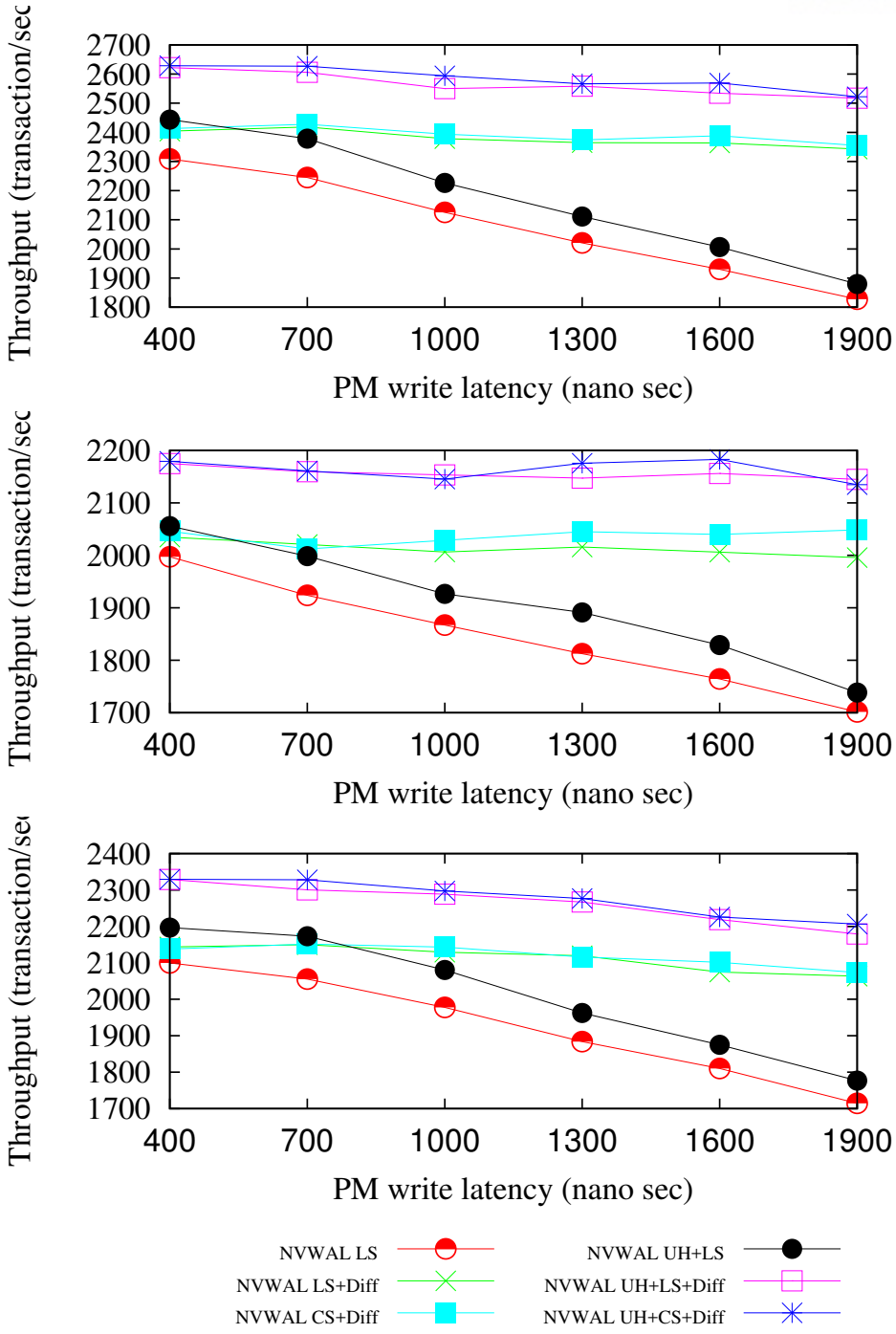


Figure 19: Transaction throughput with varying latency of Persistent Memory (Average of 5 runs on Tuna)

because it has to shift some log entries to avoid fragmentation issues, the write-ahead logging may touch a large portion of the B-tree page. Therefore, the insert transaction gets the most performance benefit from byte-granularity logging while the performance benefits for the update and delete transactions are moderate.

For the experiments shown in Figure 19, we measure the throughput of NVWAL with the

SQLite benchmark app Mobibench [51] with varying the latency of persistent memory. Using Mobibench, we submit 1,000 transactions that insert, update, or delete a single 100-byte data record per transaction. In the experiments, we do not include the time for periodic checkpointing that sporadically flushes the logged WAL frames to the slow block device storage. It should be noted that checkpointing affects the performance of only one out of hundreds of transactions; this overhead is not relevant to the write latency of persistent memory.

The write latency of persistent memory is likely to be higher than the write latency of DRAM. For example, phase change memory, because it has access latencies in the hundreds of nanoseconds, is about 2∼5 times slower than DRAM [25, 52]. In the experiments, we vary the write latency of persistent memory from 400 nsec to 1900 nsec. Overall, as the write latency increases, the transaction throughput decreases in a linear fashion. However, the latency adjustment affects the throughput of some NVWAL schemes more severely than it does others. Due to the unavailability of persist barrier instruction, we simulate the persist barrier overhead by introducing a 1 usec delay using `nop` instructions. We also assume the PoC is the system main memory - DRAM and persistent memory. In most cases, the PoC is the system main memory and `dccmvac` flushes data to the main memory. However, ARM architecture does not prohibit the implementation of outer caches beyond the PoC. We believe the outer cache should be disabled when persistent memory is used. Otherwise, outer caches should be explicitly flushed before the persist barrier instruction is called.

NVWAL LS denotes the NVWAL scheme with *lazy synchronization*, which calls cache line flush instructions in a lazy manner without hurting the correctness of the database transactions as illustrated in Figure 16(c). *NVWAL LS* does not employ byte-granularity differential logging and user-level heap; it calls Heapo's `nvmalloc()` system call for every dirty WAL frame.

In *NVWAL LS+Diff*, SQLite performs lazy synchronization. Additionally, it adopts *byte-granularity differential logging*. When the B-tree node size is 4 Kbytes, *NVWAL LS* calls 128 cache line flush instructions, one memory barrier instruction, and one persist barrier instruction for each WAL frame. It then calls another cache line flush instruction, memory barrier instruction, and persist barrier instruction for a commit mark. However, in *NVWAL LS+Diff*, the average number of cache line flush instructions per transaction is reduced, and hence *NVWAL LS+Diff* consistently outperforms *NVWAL+LS*, yielding up to 28% higher throughput.

In *NVWAL CS+Diff*, NVWAL asynchronously flushes log entries without explicitly calling cache line flush instructions, as illustrated in Figure 16(d). In this variant of NVWAL, we call the cache line flush instruction and memory barrier only for a commit mark and checksum bytes. In this way, we minimize the overhead of the ordering constraints. However, the performance benefit comes with a potential inconsistency.

In *NVWAL UH+LS*, NVWAL employs the user-level heap, which reduces the number of calls to the expensive persistent memory heap manager's `nvmalloc()` function. In the experiments shown in Figure 19, we call Heapo's `nvmalloc()` and allocate 8Kbytes persistent memory block that can store two WAL frames. By saving the overhead of calling an expensive persistent mem-

ory heap manager's function, *NVWAL UH+LS* achieves a 6% performance gain over *NVWAL LS*.

In *NVWAL UH+LS+Diff*, NVWAL employs user-level heap, lazy synchronization, and byte-granularity differential logging. Interestingly, the performance of *NVWAL UH+LS+Diff* is comparable to that of *NVWAL UH+CS+Diff*, which uses an user-level heap, byte granularity differential logging, and checksum bytes. Because *NVWAL UH+CS+Diff* minimizes the number of bytes to be written to persistent memory and also minimizes the overhead of the `cache_line_flush()` system calls and memory barrier, it shows the highest transaction throughput. Considering that *NVWAL UH+CS+Diff* is vulnerable to the inconsistency problem that is inherent in probabilistic checksum bytes, *NVWAL UH+LS+Diff*'s comparable performance makes it a promising write-ahead logging scheme for persistent memory, because it does not compromise the correctness of database transactions.

As the write latency of persistent memory increases, the benefit of using the proposed schemes becomes more significant. When the write latency of persistent memory is set to 1942 nsec, *NVWAL UH+LS+Diff* yields throughput up to 37% higher than that of *NVWAL LS*.

Experiments on Nexus 5

In the last set of experiments, we examine the performance of NVWAL using a commercially available smartphone - the Nexus 5, which has a 2.26 GHz Snapdragon 800 processor, 2 GB DDR memory and 16 GB SanDisk iNAND eMMC 4.51 SDIN8DE4 formatted with an EXT4 file system. As for the persistent memory emulation on Nexus 5, we assume that a specific address range of DRAM is persistent memory, i.e., that persistent memory is attached to the memory bus. persistent memory write latency is varied by inserting `nop` instructions. The stock SQLite WAL implementation does not use the emulated persistent memory but stores log pages in flash memory instead. We fix the CPU frequency to the maximum 2.26 GHz to reduce the standard deviation of the experiments. The Snapdragon 800 processor's cache line size is 64 bytes. We run Mobibench using SQLite 3.7.11 and Android 4.4 on Nexus 5.

Recent studies [7, 8, 18] have reported that SQLite on flash memory suffers from unexpected I/O volume and that SQLite write-ahead logging unnecessarily doubles the I/O traffic in the current implementation (SQLite 3.7.11). In order to fix these problems and compare the performance of SQLite WAL on flash memory against that on NVWAL in a fair way, we implemented two optimizations that help avoid the unnecessary I/O traffic and that significantly improve the performance of SQLite on flash memory.

For each dirty page, SQLite creates a 24-byte frame header that consists of page number, commit mark, checksum values, etc., and that appends the dirty page to it. Due to the additional 24-byte frame header of the WAL frames, each WAL frame becomes larger than the page size, which causes the WAL frames not to be aligned with the page boundaries. With such misaligned pages, a write operation for a single database page causes at least two pages to be written to the block device storage. In order to resolve the misaligned WAL frame problem, we modified

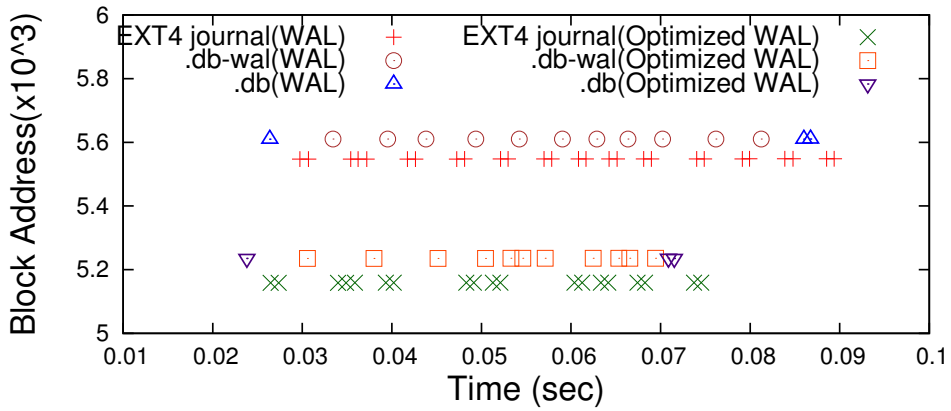


Figure 20: Block Trace of SQLite Insert Transaction with Optimizations

SQLite's B-tree split algorithm so that it splits an overflow page early and the last 24 bytes of all B-tree pages are not used. By doing so, 24 bytes of WAL frame header and WAL frame can be merged and stored in a single page of a WAL file. We implemented the same split algorithm for NVWAL.

Another ad-hoc improvement we made on SQLite WAL is *pre-allocation of WAL log pages* as in WALDIO [18]. For each dirty page, a transaction calls `write()` that appends one or two pages to the end of a log file. As the appended pages increase the size of the WAL log file, the increased file size must be updated in an inode. Because the writes in WAL mode are all sequential and because the size of the log file keeps increasing until checkpointing truncates it, pre-allocating multiple pages once will help the next transactions write log frames without increasing the file size. The only negative aspect of this pre-allocation is that it may waste several disk pages if there is no next transaction. In terms of EXT4 journal overhead, allocating two new pages causes overhead slightly higher than that induced by allocating only one page. However, that overhead is much smaller than the overhead for allocating another page later. The size of the pre-allocated pages can be fixed at a specific number based on the database access patterns or the size can be doubled every time the pre-allocated pages fill up.

Figure 20 shows a block I/O trace of 10 transactions in stock SQLite WAL mode and another block I/O trace of 10 transactions in our optimized WAL mode. For a single insert transaction, one block (4KB) is written to the `.db-wal` file but two blocks(16KB, 4KB) are written to the EXT4 journal to update the metadata of the log file in the ordered-mode EXT4 journal. In our optimized WAL mode, SQLite pre-allocates 8 pages when a transaction writes a log frame for the first time, even if the system needs just one page. If another transaction needs to write log frames while there is no available pre-allocated page, we double the number of pages to be pre-allocated each time and SQLite allocates 16 new pages to the log file. Compared to the block trace of the WAL mode, allocating multiple log pages in advance reduces EXT4 journal accesses by 40% (172 KB vs. 284 KB), and the batch execution time in our optimized WAL mode decreases from 90 msec to 74 msec.

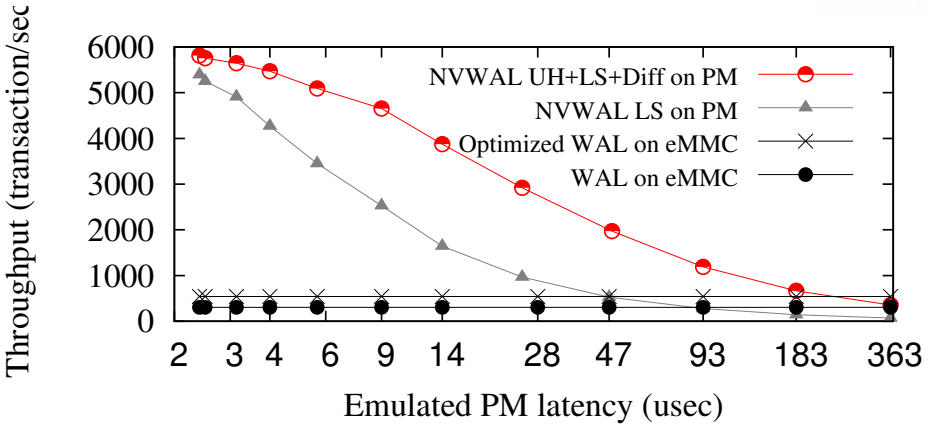


Figure 21: Transaction Throughput of NVWAL on Emulated Persistent Memory vs. Optimized WAL on eMMC Memory (Average of 5 Runs)

In Figure 21, we measure the transaction throughput of NVWAL on Nexus 5. We run 1000 insert transactions with an empty SQLite database table. Each transaction inserts a single 100-byte record into the table. The persistent memory write latency on the Nexus 5 is varied by adding `nop` instructions after each `clflush` instruction, which is also used to emulate the persistent memory write latency in [5]. We set the checkpointing interval to 1000 dirty WAL frames, which is the default setting in SQLite. As for the sporadic checkpointing overhead, we amortize the overhead across 1000 transactions.

Our optimized WAL on flash memory yields 541 transactions/sec while *NVWAL LS* and *NVWAL UH+LS+Diff* on persistent memory (DRAM with 2 usec write latency) exhibit 5393 and 5812 transactions/sec, respectively. As we increase the write latency, the throughput of NVWAL decreases. When the write latency becomes 47 usec, *NVWAL LS* on the emulated persistent memory shows throughput similar to that of WAL on flash memory. When the write latency is set to 230 usec, *NVWAL UH+LS+Diff* also shows performance similar to that of WAL on flash memory. We need to stress that an persistent memory write latency of 230 usec is very conservative.

The experiments on the Nexus 5 are not precisely identical to the experiments on the Tuna board because the former do not include the overhead of checkpoint operation, whereas the latter include the checkpointing overhead. Hence, the experiments on the Nexus 5 stand for the sustained throughput, while the experiments on the Tuna stand for the peak throughput.

4.6 Failure-Atomic Slotted-Paging[5]

Although NVWAL [2] successfully reduced the amount of I/O for data consistency guarantee, There are still large number of write amplification from the redundant I/O for memory copy from volatile memory to persistent memory. To eliminate this redundant I/O, we use the persistent

[5]This work was published in proceedings of ASPLOS'17 and the contents of this work are also included in Jihye Seo's thesis for her Master degree.

memory as persistent buffer cache. In this section, we will discuss how to fully exploit the persistent buffer cache with the slotted page structure that is commonly used in commercial database systems.

The *slotted-page structure* is commonly used for organizing variable-length records in a fixed-sized block [1, 53, 54]. In a slotted-page structure, there is a *slot-header* at the beginning of each page, *free space* in the middle, and a *record content area* at the end of the page. The slot-header contains metadata about each page, including the number of records in the page, the end of free space in the page (the beginning of record content area), and an array whose entries contain the location of each record, which we refer to as *record offset array*

When a new record is inserted into a slotted-page, space is allocated for the record at the end of the free space extending the record content area. Note that the record content area also contains the length of the record and that it grows towards the beginning of the page. The offset and size of the new space allocated is added to the record offset array of the slot-header. The record offset array grows towards the end of the page. As the new records are always placed at the head of the record content area regardless of their keys, the record offset array is always kept sorted according to the ordering of the keys in B+-tree file format.

Slotted-Paging and In-Place Commit

The slotted-page structure was originally designed for block device storage. When a transaction makes changes to a slotted-page, it copies the 4K or 8K byte sized page from the block storage device to volatile main memory (the buffer cache), updates the entire page, and flushes the page to the block storage device.

If we replace volatile main memory with PM, a transaction's commit operation does not have to flush updated slotted-pages to slow block storage. Instead, we can simply place a commit mark for any change that has taken place in the slotted-page. Such a scheme eliminates unnecessary redundant writes minimizing I/O and memory operations, which will help improve the performance of transactions and mitigate the endurance problems of PM.

The key idea behind the in-place commit scheme is to ensure that all modifications become committed and viewable with a single failure-atomic write operation. Let us see how this is done with persistent slotted-paging, that is, a slotted-page in persistent memory.

Failure-atomic write instructions are expected to be supported at 8 bytes or a larger granularity for PM [25, 28], and hardware transactional memory is one way of achieving the same goal. The advent of commercially available hardware transactional memory such as the Intel's Restricted Transactional Memory (RTM) and Hardware Lock Elision (HLE) has opened up a new means to support coarse-grained atomic operations via hardware support. Hardware transactional memory has spurred the growth of interest in using hardware-assisted transactional support in database systems [42, 55, 56]. In this work, we employ RTM as we need a user-defined fallback execution path if hardware transactions abort. RTM, in particular, provides three new instructions - `XBEGIN`, `XEND`, and `XABORT` that allow programmers to specify the start and end

of a transaction, and to explicitly abort a transaction if the transaction cannot be successfully executed [36, 42]. RTM guarantees that the store operations within a single transaction are not visible outside the transaction until `XEND` is successfully completed. That is, a dirty cache line remains in the *write combining store buffer*, which combines multiple consecutive small 8 bytes writes, without flushing the cache line to the memory subsystem. If the system crashes before the transaction finishes, the dirty cache line will be lost and it does not hurt the consistency of the memory subsystem.

One of the major limitations of RTM is that an RTM transaction cannot successfully commit if its working set (i.e., read and write sets) size exceeds the hardware limit. Since RTM transactions could keep getting aborted due to consecutive transactional overflows, there is no guarantee for forward progress. Even if the working set size of persistent slotted-paging is smaller than the hardware limit, multiple dirty cache lines cannot be flushed to PM atomically. Hence, as in [36], we assume that the underlying hardware supports failure atomicity at cache line granularity, and we restrict the working set size of RTM to be no larger than the cache line size. [6]

In our in-place commit scheme, we make use of hardware transactional memory as follows:

Inserting a record: Suppose a transaction inserts a record into a persistent slotted-page. The actions that are involved with insertion are (i) writing the record along with its length into the record content area and (ii) writing to the slot-header, which involves writing the offset of a new record into the record offset array, updating the number of records, and updating the start of the record content area. Note that writing to the record content area need not be atomic as other transactions cannot see this addition until the slot-header is updated. So, if we can assure that the slot-header is no larger than the cache line size and failure-atomicity is ensured for a cache line write - (ii), then the insertion can be done atomically. [7] This is the premise behind our proposed in-place commit.

Hence, after writing the record, we update the slot-header in the RTM region by inserting the new record offset into its record offset array and increasing the number of records. Since the slot-header in the store buffer can be atomically written to PM by a cache line flush instruction, the slot-header acts as an in-place commit mark ensuring that all actions within the insert transaction has been committed and that the entire slotted-page will always be consistent, durable and fail-safe. Note that if a system crashes while adding a new record into the record content area, the partially written slot-header is, at best, in the RTM region. Thus, the slot-header in PM is not altered and the partially written data will simply be ignored.

[6] Nevertheless, best-effort HTM transactions are not guaranteed to succeed. Hence, if an RTM transaction fails, our fallback handler retries the RTM transaction until it succeeds. Alternatively, we can implement a handler that falls back to slot-header logging that we describe in section 4.6 if RTM transactions continuously fail.

[7] We use RTM for atomicity and consistency, but not for durability and isolation. RTM is used just to prevent a partially updated cache line from being written to PM. Durability is guaranteed when we call `clflush` after the RTM transaction ends. It should be noted that `clflush` cannot be called inside the RTM region since it violates the property of hardware transactions.

Updating a record: When a record is updated in a persistent slotted-page, the record should not be overwritten for recovery purpose. Hence, we add an updated record in free space and atomically replace the offset of the previous record in the record offset array with the new offset so that the previous record is marked as deleted and the newly added record becomes accessible.

Deleting a record: If a transaction deletes a record, we can atomically invalidate the record by deleting its offset from the record offset array and decreasing the number of records in the slot-header via RTM transaction.

These insert/update/delete operations in a persistent slotted-page structure guarantee failure atomicity of a single slotted-page, i.e., a slotted-page is atomically transformed from one consistent state to another consistent state even if a system crashes.

In Android applications, it is known that most write transactions insert just a single data item into the SQLite database as if it is a flat file interface [8]. For such single write transactions, the in-place commit scheme in persistent slotted-page structure is optimal in the sense that it minimizes memory write operations as it does not create any copy page. It requires a minimal number of `store` and `clflush` instructions for the dirty record in the record content area and only one `store` instruction and one `clflush` instruction for the commit mark.

Slot-Header Logging

The in-place commit scheme and cache line atomic write granularity is, unfortunately, far from satisfactory when a slotted-page structure needs to split or when a transaction updates multiple pages. Splitting a slotted-page must atomically update the slot-headers of two pages - the page that splits and its parent page. Even with RTM, two slot-headers cannot be updated atomically because the underlying PM does not support failure atomicity of writing two separate cache lines. Moreover, database transactions that insert more than one record are not uncommon in enterprise database systems. If they modify multiple pages, the in-place commit scheme alone cannot guarantee failure atomicity of transactions.

In order to provide failure atomicity for a transaction that modifies multiple pages, we have no choice but to fall back to logging methods. However, notice here that now as the database buffer cache is non-volatile, there is no reason to duplicate the dirty bytes in the journal or log because they are already persistent in the buffer cache. In the *slot-header logging* scheme that we propose, we do not duplicate the records in the record content area but do so only for the slot-header, that is, we write only the small slot-header in a separate persistent log, so that we minimize the memory write operations. Since the slot-header in the slotted-page structure behaves as a per-page commit mark, we write the slot-header in a separate log and postpone applying the log to the actual pages until the entire transaction is ready to commit.

If a transaction modifies page A and page B, we perform in-place updates in the record content area of page A and B, but the slot-headers are not updated in-place. In slot-header logging, the ordering of memory write operations for the two pages does not have to be enforced

as long as they are flushed to PM before we write the slot-headers to the log.

After calling cache line flush instructions and memory barrier instructions for the in-place updates in the record content area, we copy and update the slot headers (record offset arrays) of the two pages in a separate log, which we refer to as *slot-header log*. Since we do not overwrite the slot-headers of pages A and B in-place, the updated slot-headers in the slot-header log does not have to be atomically written. Furthermore, the slot-headers of pages A and B do not have to be stored in a specific order as long as they are flushed to PM before their transaction commit mark is written.

After the transaction commits and its commit mark is flushed to the slot-header log, we immediately start checkpointing the slot-header of each page from the slot-header log to the actual pages A and B. This is unlike legacy checkpointing where we generally postpone checkpointing until on opportune time. This eager checkpointing is done so that other transactions do not have to check the slot-header log. Once checkpointing is done, the updated records in the pages become accessible by other transactions, and we can safely remove the slot-header log. This is a feasible approach as our slot-header logging is lightweight since each log frame per slotted-page is just the metadata of each page and writes are happening in PM. In contrast, in legacy write-ahead logging, checkpointing is a very expensive operation because all dirty pages are being flushed to slow block device storage.

4.7 Summary

Emerging non-volatile memory devices are expected to substitute slow block device storage so as to persistently store frequently accessed data. In this work, we design and implement write-ahead-logging on persistent memory (NVWAL) and show that NVWAL can minimize the overhead of managing and synchronizing log entries on persistent memory via i) transactionaware lazy synchronization, ii) a user-level persistent memory block manager, and iii) byte-granularity differential logging. Through our extensive performance evaluation, we show that database logging can be optimized to be insensitive to persistent memory latency, and that the overhead of guaranteeing the failure atomicity is almost free. NVWAL with the optimizations exhibits transaction throughput up to 37% higher than that of non-optimized WAL on persistent memory; when persistent memory latency is smaller than 2 usec, NVWAL achieves at least 10x higher than that of legacy WAL on flash memory. We can draw three lessons from this study. First, employing an persistent memory in database logging is not an option but a necessity. Second, the aggregate overhead of the persist barrier instructions is insignificant from the application's point of view. Third, persistent memory latency is barely propagated to SQLite performance. NVWAL significantly decreases the logging overhead in SQLite and makes the workload further CPU bound. As a result, SQLite performance becomes relatively insensitive to persistent memory latency.

Moreover, we developed failure-atomic slotted paging with persistent memory as a persistent buffer cache to minimize the redundant I/O. With failure-atomic slotted paging, we can

guarantee the data consistency without data copy.

V Byte-Addressable Persistent Index

Recent advances of byte-addressable persistent memories, such as phase change memory and spin-transfer torque RAM, open up a new opportunity to reduce the I/O overhead of block granularity storage systems [8, 18]. However, the advent of non-volatile memory places new challenges on existing software including B-trees, which are designed for volatile memories and block device storage systems.

One of the biggest challenges of page-based data structures for persistent meomory is how to atomically flush an updated page because persistent memory is expected to have small granularity of failure-atomic writes - 8 bytes [25,28]. In particular, there is no guarantee of when the updates to a page will be written back to the persistent memory because they can be cached in L1 and L2 caches and the memory controller can reorder writes at a cache line granularity in order to fully utilize the memory bus bandwidth. Although the ordering of memory writes can be preserved via memory fence and cache line flush instructions, calling a large number of cache line flush instructions for a small change in tree nodes, as was done in CDDS B-tree [9], can degrade the insertion performance since the cache line flush is known to be very expensive [5, 10, 16]. In order to reduce the overhead of cache line flush, Intel recently added various options for persist instructions including `clwb`, which does not throw away the flushed cache line from CPU cache. However, still cache line flush makes write operations sensitive to the latency of persistent memory, while other `store` instructions can hide the persistent memory write latency by leveraging large CPU cache.

In this work, we design and implement a variant of a B-tree optimized for a small cache line - *clfB-tree* (Cache Line-Friendly B-tree). The key ingredients of clfB-tree are four fold - i) *cache line fit node structure* that needs just a single cache line flush, ii) *delta encoding compression* that increases the degree of a small node, iii) *failure-atomic shift* (FAST) for key sorting in a single tree node, and iv) *failure-atomic in-place rebalancing* (FAIR) node split/merge that eliminates the necessity of logging.

5.1 Cache Line-Friendly B-tree

Delta Encoding

If a tree node is small, the node will more frequently overflow and split than larger tree nodes. Whenever a tree node splits, we need to allocate a new memory block from the memory pool; this process is managed by memory allocators such as `ptmalloc2`, `dlmalloc`, `jemalloc`, `tcmalloc`, etc. These memory allocators internally manage the memory pool in such a way that consecutive `malloc()` requests of small chunks end up being allocated close to each other. For example, Google's `tcmalloc` assigns each thread a thread-local cache. Such a memory allocation policy yields a system in which the memory addresses in an index node have high locality.

In order to take the advantage of this memory locality, we develop *delta encoding* for Cache

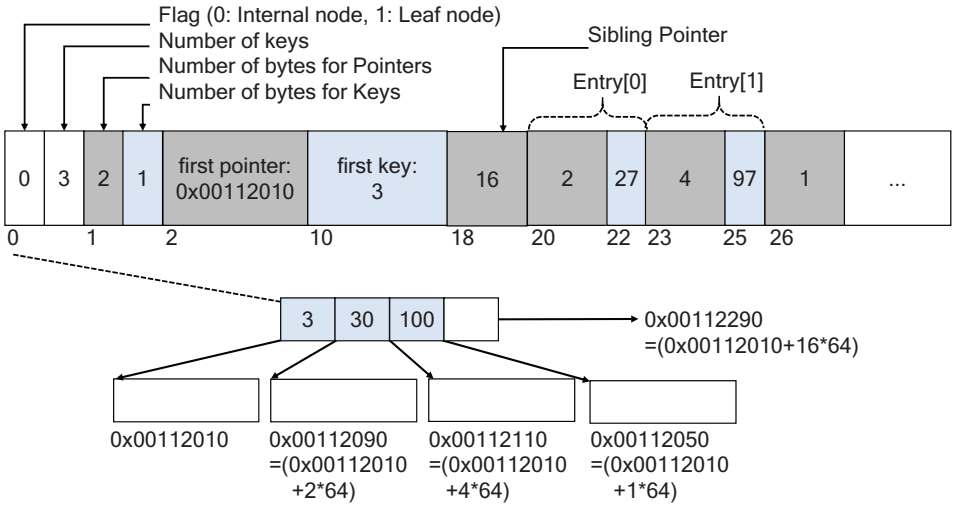


Figure 22: *Layout of clfB-Tree Node with Delta Encoding*

Line-Friendly B-tree. In our delta encoding process, the first pointer in a tree node is stored as it is. From the second pointer, we compute the *delta* from the first pointer ($second\_pointer - first\_pointer$) and divide it by the cache line size to make the delta even smaller. Note that clfB-tree allocates cache line aligned memory blocks for each tree node. Then, we compute how many bytes are needed to store the delta. For example, if the delta is less than 127 but greater than or equal to -128, we store the delta in just one byte. If the delta is smaller than 32,767 but greater than or equal to -32,768, we store the delta in two bytes. As such, we encode keys and pointers into 1, 2, 4, or 8 bytes space. As the memory allocation locality becomes higher, the delta encoding compression can store more pointers.

As for the keys, indexing structures that store key-value pairs in a sorted order also have high locality. Let's suppose a tree node stores three keys of 8 bytes - 123000, 123010, and 123020. The smallest key 12300 is stored in the node as it is. However, our delta encoding scheme stores the delta, i.e., 10 in one byte and 20 in another one byte. Thereby, we can save a total of 14 bytes in the index node.

Although the delta encoding can save more space if we encode keys and pointers in 3, 5, 6, or 7 bytes, there is a trade-off between performance and space saving. That is, these are not the sizes of supported data types. And we have observed that the encoding/decoding overhead of such non supported data type is not ignorable. Hence we encode keys and pointers only in 1, 2, 4, and 8 byte data types.

Node Structure

Figure 22 shows the layout of a clfB-tree node. The size of a clfB-tree node is set to 64 bytes in the example. The first two bytes store metadata about the tree node. The first bit indicates whether the tree node is a leaf or an internal node and the rest of the bits in the first byte represents how many keys are in the tree node. The second byte represents how many bytes are

used for encoded pointers and keys. After the two-byte metadata, we store the 8-byte pointer to the leftmost child node and a sibling pointer next to it. We defer our discussion of this sibling pointer, which is used for the node split algorithm, until section 5.3. Next to the sibling pointer, we store the rest of keys and pointers. Except for the first key and first pointer, we store the encoded keys and pointers using the delta encoding scheme we described in section 5.1.

If the pointer encoding and the key encoding require 1 byte respectively, a 64 byte tree node can have a maximum of 23 keys and 24 child pointers. We can further compress the keys, pointers, and metadata to the unit of a bit, but we found that bitwise encoding requires more computations and it slows down the overall indexing performance.

Suppose a clfB-tree node stores three keys - 3, 30, and 100, and four child pointers, as illustrated in Figure 22. The 8 bytes pointer to the leftmost child node is stored after the two bytes metadata, and the smallest key, 3, is stored next to it. For the rest of the pointers and the keys, we compute the differences against the first value and determine how many bytes are required to store the maximum difference. As for the keys, the values monotonically increase, and thus we can simply calculate and encode the difference between the first key and the last key. For the pointers, the memory addresses are not necessarily in an increasing order. Thus, we use the most significant bit as a sign bit for the encoding.

Delta Decoding and Search

Although the keys in clfB-tree node are encoded, we do not have to decode all keys for comparison. As shown in Algorithm 6, we encode the search key in each tree node and compares it against the encoded keys. Since the encoding preserves the ordering of keys in its encoded form, we do not need to decode the keys. In this way, we can minimize the decoding overhead.

Restricted Transactional Memory

Failure-atomic write instructions are expected to be supported at a higher granularity than 8 bytes for persistent memory [25,28], and hardware transactional memory is one way of achieving the goal. The advent of Intel's restricted transactional memory (RTM) supports coarse-grained atomic operations via hardware support. RTM has spurred the growth of interest in using hardware-assisted transactional support in various domains [36, 42, 55, 56].

RTM guarantees that the store operations within a single transaction are not visible outside the transaction until `XEND` is successfully completed. That is, a dirty cache line remains in the *write combining store buffer*, which combines multiple consecutive small 8 bytes writes, without flushing the cache line to the memory subsystem. If the system crashes before the transaction finishes, the dirty cache line will be lost and it does not hurt the consistency of the memory subsystem.

One of the major limitations of RTM is that an RTM transaction cannot successfully commit if its working set (i.e., read and write sets) size exceeds the hardware limit. However, since the

Algorithm 6: $Search(node, key)$

procedure

```
1: key_delta ← key − node.first_key;
2: if node.isLeaf() then
3:    for i ← 0, node.numEntries − 1 do
4:       if key_delta == node.getEncKey(i) then
5:          return node.getEncPtr(i) + node.first_ptr;
6:       else if key_delta < node.getEncKey(i) then
7:          return;
8:       end if
9:    end for
10:   if node.sbl_ptr != NULL then
11:      sibling_node ← node.sbl_ptr;
12:      Search(sibling_node, key);
13:   end if
14: else
15:   for i ← 0, node.numEntries − 1 do
16:      if key_delta <= node.getEncKey(i) then
17:         search(node.getEncPtr(i) + node.first_ptr, key);
18:      end if
19:   end for
20: end if
21: // fail to search
22: return NULL;
```

end procedure

size of clfB-tree node is a single cache line, RTM transactions are guaranteed for forward progress.

5.2 FAST: Failure-Atomic ShifT

FAST Insertion

Using RTM, clfB-tree can atomically update an entire tree node. With the RTM, it is trivial to insert a new key and its child pointer into a clfB-tree node atomically. However, not all the processors support the hardware transactional memory extension. Also, it is easy to guarantee the consistency of clfB-tree node simply by taking advantage of the property of B+-tree - sorted keys.

Since the keys in a B+-tree node are sorted, we need to shift some keys and pointers to the right to make a space for a new entry. While we are performing these shift operations, the system may crash. After the system recovers, the tree node can have duplicate keys and pointers in the middle as shown in Figure 23, which seems at first glance to be inconsistent.

However, the duplicate keys and pointers in B+-tree node can be ignored as long as we do

Algorithm 7: $Insert(node, key, ptr)$

procedure

```
1: if node.cnt < node_capacity then
2:    key_delta ← key − node.first_key
3:    node.getEncPtr(node.cnt) ←
      node.getEncPtr(node.cnt − 1);
4:    node.getEncKey(node.cnt − 1) ←
      node.getEncKey(node.cnt − 2);
5:    mfence();
6:    node.cnt ← node.cnt + 1;
7:    for i ← node.cnt − 2; i >= 0; i++ do
8:       if key_delta < node.getPtr(i) then
9:          // shift to right
10:         node.getEncPtr(i + 1) ← node.getEncPtr(i);
11:         node.getEncKey(i) ← node.getEncKey(i − 1);
12:      else
13:         node.storeEncPtr(i + 1, ptr − node.first_ptr);
14:         node.storeEncKey(i, key_delta);
15:         break;
16:      end if
17:   end for
18:   mfence();
19:   clflush(node);
20:   mfence();
21: else
22:   split(node);
23: end if
```

end procedure

not allow duplicate keys in the tree nodes. If a dataset has more than one data that has the same search key, it can be handled by storing the pointer to a bucket that contains multiple records with the same key. Another alternative and simple solution, used by most database systems, is to make the search key unique by adding another attribute to the record, such as record-id [1]. Duplicate keys, especially in internal nodes, have no meaning in indexing structures because they fail to distinguish child nodes. As such, clfB+-tree node guarantees $K_0 < K_1 < ... < K_{n-2} < K_{n-1}$, and we can easily detect whether a child pointer is valid or not. If there are two identical keys K_i, K_{i+1}, the child pointer in between the two keys should be ignored or recovered as we will describe later.

For example, suppose we insert a key, 40, into a tree node, as shown in Figure 23. For simplicity, we do not show the delta encoding in the figure. In order to make a space for 40, 50 and 60 and their pointers need to be shifted to the right. We first shift 60's right child pointer

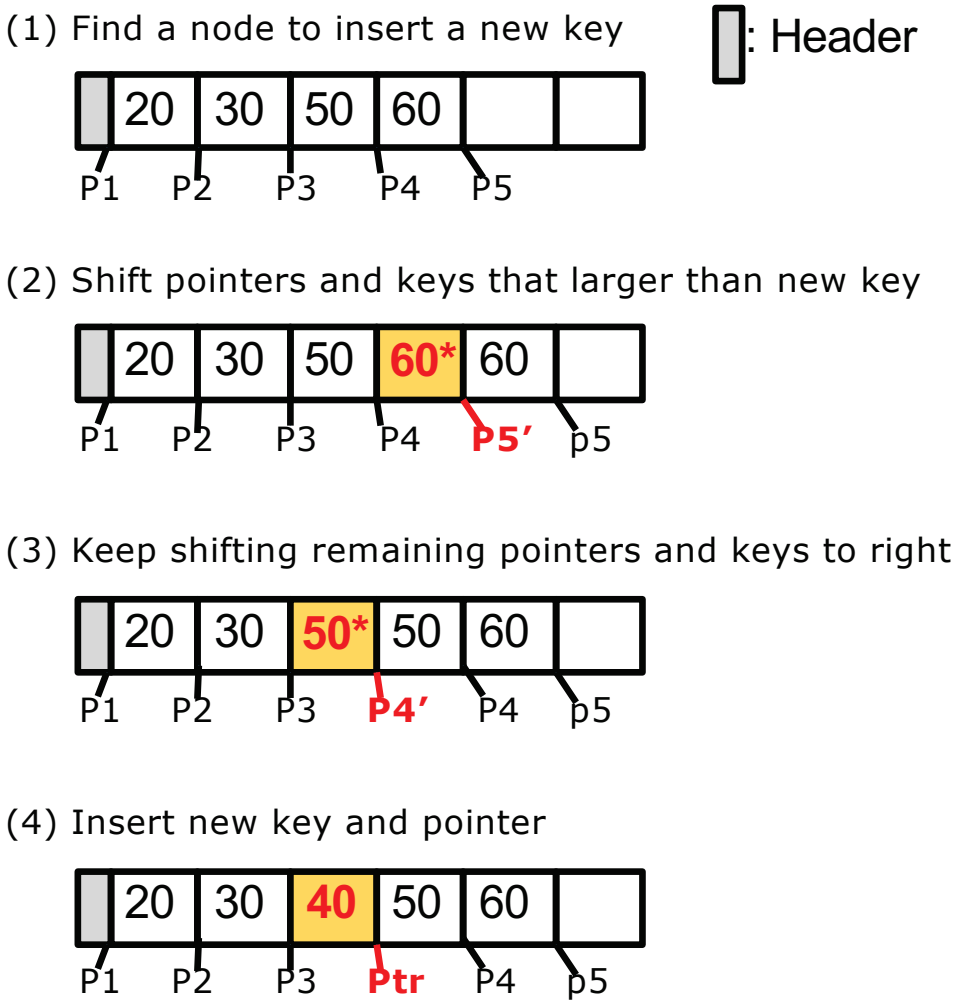


Figure 23: *Insertion into clfB-tree node*

to the right, and then key 60. If the system crashes in step (2), we will have duplicate keys and pointers - two 60s and two $P5$s. When accessing this node, it is easy to detect the duplicate keys, and we can ignore $P5'$ in the middle of the duplicate keys.

Similarly, in the next step, we overwrite $P5'$ with $P4$. Even if the system crashes at that point, the new $P4$ will still be ignored since it is in between duplicate keys 60 and 60. After that, we shift 50 to make a space for 40, as shown in step (3). If the system crashes at this point, $P4'$ will still be ignored because it is in between duplicate keys 50 and 50. Finally, in step (4), we write a new child pointer Ptr first, and then 40 will be inserted as the third key. Writing the new key now validates the child pointer Ptr, i.e., the key behaves as a commit mark of the child pointer.

The 8-byte write operations in FAST algorithm, shown in 7 are all failure-atomic in a sense that *failure atomicity* means that updated data should be still usable even if a system crashes.

These failure-atomic write operations must be performed in this particular order. In modern processors, memory access instructions are often reordered to maximize the memory bandwidth

utilization. However, if store instructions have dependencies, processors do not reorder them. Since the store instructions in FAST are all dependent, we do not need to call any memory fence instructions explicitly except the first shift operation - the largest key. The largest key and its right child pointer must be written before we increase the number of entries in the tree node. Writing to the number of entries field has no dependency with the shift operations. Hence, if we do not call `mfence` instruction, processors can first increase the number of entries field before we shift the largest key and pointer. If we increase the number of entries before the first shift and if the system crashes, the garbage key and pointer can be accessed by other queries.

However, we do not need to call cache line flush instruction after increasing the number of entries field because the keys and pointers are all in the same cache line with the number of entries field in clfB-tree node. While shifting keys and pointers, the cache line can be flushed to persistent memory if CPU cache replacement occurs. However, still the ordering of these write operations will be preserved, and the partially written tree nodes do not hurt the consistency of tree nodes.

Only after we write a new child pointer and a new key do we need to call `mfence` and `clflush`, and `mfence` instructions. Note that we need to call `mfence` instruction because `clflush` instruction is ordered only with respect to `mfence` instruction.

FAST Deletion

Deletion of a key in a clfB+-tree node can be done in a similar way. We shift the keys to the left if they are greater than the deleted key. The deletion algorithm can be illustrated by the example shown in Figure 23, following the reverse order. In order to delete 40 in the example, we shift 50 to the left. Note that, again, this shift is an atomic operation, and it behaves as the commit mark of the deletion. If the system crashes at this point, the undeleted pointer Ptr will be ignored because it is between 50 and 50. Since then, we shift $P4$, 60, and $P5$ to the left. System failures during this shift operation are tolerable, and do not need immediate recovery process. The recovery can be done in a lazy manner when we need to write the node for a subsequent update operation.

5.3 Logless Node Split and Merge

Failure-Atomic In-place Rebalancing (FAIR)
Node Split

In legacy disk-based B-tree variants, logging or journaling has been used to split tree nodes. This is because tree node split modifies multiple tree nodes, i.e., i) creates a sibling node, ii) overwrites the overflown node (in-place update) or creates an updated overflown node (CoW update), and iii) inserts a new child pointer(s) to their parent node. Since these three steps should be atomically performed, logging or journaling can be used to guarantee the consistency and the failure atomicity. However, the legacy logging and journaling create copies of the

modified pages first in volatile main memory and then in persistent block device storage as log entries. After the logging is done, the updated nodes in the log are copied to the original tree nodes. Such redundant write operations have been pointed out as a major source of performance degradation especially for mobile database systems [7, 8].

In this work, we show how we can avoid expensive logging by leveraging the property of sorted keys in B+-tree nodes. Figure 24 illustrates each step of our *Failure-Atomic In-place Rebalancing* (FAIR) node split algorithm. First, suppose there is only one leaf node in the tree as shown in (1). If we insert a new key 25, this leaf node A will split. In our clfB-tree node, we store a sibling pointer not just for leaf nodes but also for internal nodes.

As shown in (2), we create a sibling node B and set the sibling pointer point to the address of B. It should be noted that, the sibling pointer must be written before we delete the migrated entries in the overflown node A. It looks as if the consistency of the tree nodes is violated because we have duplicate entries in A and B; however, the right sibling node B will not be used because the smallest key in the right sibling node is greater than the largest key in the overflown node. As a walking example, consider the system failure in state (2). If a query 35 is submitted, the query will access the node A following the root pointer. Since 35 is smaller than 40 (the one in node A), the search will stop without accessing node B. If a query 45, that is greater than the largest key in the overflown node A, is submitted, the query should access the sibling node B and compare to see if the next key is greater than the previous key. If it is, the query will keep checking the next key until it finds a matching key or a larger key than the search key. But in this example, the search will stop after checking the first key 30 in the sibling node B.

The basic assumption that makes this redundancy and inconsistency tolerable is that right sibling nodes always have larger keys than left sibling nodes. Therefore, the keys and pointers in the new right sibling node ($< 30*, P3' >, < 40*, P4' >$) will not be valid until we delete them in the overflown node A.

For node split, it is trivial to delete the migrated keys and pointers from the overflown node. We can simply decrease the number of entries in the header using an atomic 8-byte write operation. This atomic write will change the tree status to (3). If we search 30 after deleting the migrated entries, the query will find $< 30, P3 >$ in node B after following the sibling pointer.

After we successfully split the overflown node, we need to create a new root node C as shown in (4). Creating a new root node does not require any specific ordering since it will not be accessed until we update the root pointer. If a query is submitted before we update the root pointer, the query will access the old root node A and the sibling node B as in step (3).

Once we update the root pointer as shown in (5), the clfB-tree is now in complete state. Suppose we insert more entries into node A and it overflows again. As in step (2), we create a sibling node D and migrate half entries to it as shown in (6). As in the linked list, we set the sibling pointer of node D to the address of node B, and then the sibling pointer of node A to the address of node D. Again, half of the entries in the overflown node A are redundant. But,

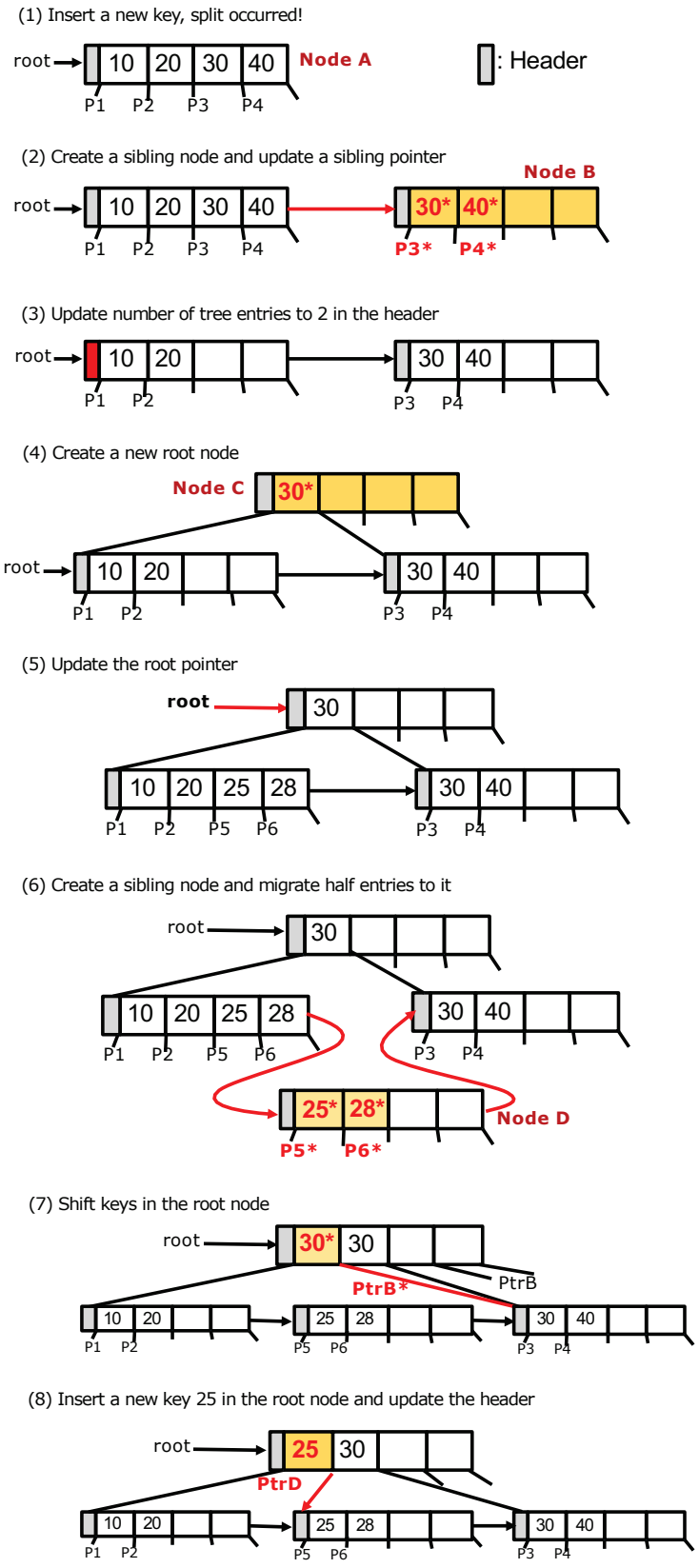


Figure 24: Failure-Atomic In-place Rebalancing

the duplicate entries do not hurt the failure atomicity of tree structure.

We now insert a new entry for Node D in the root node C. This step requires the system to shift existing entries in node C to the right, which can be performed in a failure-atomic fashion using FAST. Suppose a system crashes in (7). After system restarts, queries that search smaller keys than 30 will be forwarded to Node A. If a query is looking for 28, the query will follow the sibling pointer of Node A to find 28 in the node D. If queries that search larger than 30 access the root node C, they will follow pointer $PtrB$ instead of $PtrB*$ to access node B.

Once we make a space in the root node, we add a pointer to Node D first, and then its smallest key ($PtrD$ and 25 in the example) as shown in (8). This state makes the clfB-tree complete and the insertion process finishes.

As we described, the FAIR node split algorithm does not need expensive logging and redundant writes. Hence, it minimizes the write operation and cache line flush operations. It should be noted that FAIR node split in clfB-tree requires only three cache line flush instructions. The detail algorithm is described in Algorithm 8.

FAIR node split requires that the sibling pointer must be updated before we decrease the number of entries in the overflown tree node (node A in Figure 24 example). Otherwise the consistency is not guaranteed because the sibling pointer and the number of entries are not dependent updates from the perspective of processors. If the sibling pointer and the number of entries are in different cache line, we must call a `clflush` before we update the number of entries because it is not certain guaranteed which cache line will be flushed first if we do not explicitly call `clflush`. However, since the sibling pointer and the number of entries are in the same cache line in clfB-tree, we can enforce the ordering of memory writes via a less expensive `mfence` instruction. Because we do not explicitly call `clflush`, the updated sibling pointer can reside in CPU cache. Once, the sibling pointer is written to CPU cache, the sibling pointer is guaranteed to be flushed to persistent memory before or together with the number of entries. As we described, even if the sibling pointer is written back to persistent memory. before the number of entries is updated, it does not hurt the consistency because the partially written tree nodes guarantee the failure atomicity as we discussed above.

With FAIR node split algorithm, we do not need instant recovery process because the partially updated tree nodes are all reusable. Hence, we trigger lazy recovery process only when we find duplicate keys in tree nodes.

FAIR Node Merge

In clfB-tree, underutilized nodes can be merged in the reverse order of the split algorithm. Suppose the node D in the example shown in Figure 24 (8) is underutilized and we merge it with its left sibling node A. As can be seen in (7), we shift the keys in the parent node to the left so that we delete the index key of the underutilized node. The $PtrD$ will be in the middle of the duplicate keys, and thus it will not be used.

In the next step, we copy the keys from underutilized node D to the left sibling node, which

Algorithm 8:
FAIRsplit(node)
procedure

```
1: sibling ← nv_malloc(sizeof(node));
2: parent ← path_stack(top − −);
3: if node.isLeaf() then
4:    for i = capacity/2; i < capacity; i + + do
5:       sibling.store(node.getEncKey(i),
         node.getEncPtr(i + 1));
6:    end for
7: else
8:    sibling.first_ptr ← node.getEncPtr(capacity/2) + node.first_ptr;
9:    for i = capacity/2 + 1; i < capacity; i + + do
10:      sibling.store(node.getEncKey(i),
         node.getEncPtr(i + 1));
11:   end for
12: end if
13: mfence();
14: clflush(sibling);
15: mfence();
16: node.sibling_ptr ← node.first_ptr − sibling;
17: mfence();
18: if node.isLeaf() then
19:   node.cnt− = sibling.cnt;
20: else
21:   node.cnt− = (sibling.cnt + 1);
22: end if
23: mfence();
24: clflush(node);
25: mfence();
26: parent.store(node.getEncKey(capacity/2),
    node.sibling_ptr);
```

end procedure

will make the tree structure shown in (6). After we call `mfence` for the merged entries in node A, we update the sibling pointer of node A to point to node B. Then we call `clflush`, which will cause node D be deleted from the tree structure.

5.4 Journaling for Delta Encoding

Although the failure-atomic shift insertion/deletion and FAIR node split/merge algorithm can successfully eliminates the necessity of expensive logging or journaling, there is a certain case

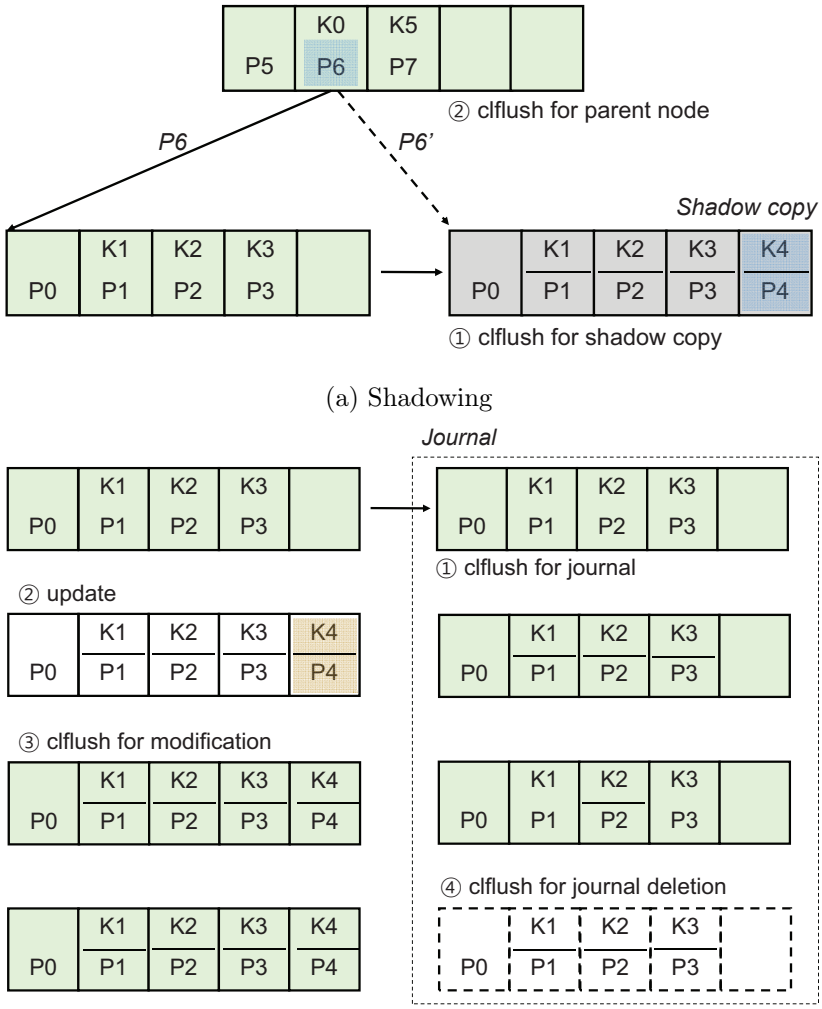


Figure 25: Journaling and Shadowing for Atomic Insertion

when FAIR is not applicable because of the delta encoding. In this section, we discuss how clfB-tree can atomically update a tree node without the proposed failure-atomic shift insertion and FAIR node split/merge algorithm.

Short-Circuit Shadowing

Shadowing is a copy-on-write technique that is widely being used to provide atomicity and recoverability in database systems. As shown in Figure 25a, (1) short-circuit shadowing [25] creates a copy of a tree node and modifies the copy. After the shadow copy is persisted, (2) the parent node replaces the pointer to the original tree node with the address of the shadow copy via an 8-byte atomic write operation.

The short-circuit shadowing in clfB-tree requires two cache line flush instructions - one for the shadow node and the other for the parent node. However, the delta encoding makes this short-circuit shadowing suffer from *upward cascading shadowing problem*. Since the address of

the shadow copy is different from that of the original tree node, the number of bytes required for the shadow copy may exceed the number of bytes used for delta encoding in the parent node. If so, the parent node needs to be updated and we create a shadow copy of the parent node.

Unfortunately, the short-circuit shadowing in clfB-tree may result in cascading split problems because the pointer, in the parent node, to the shadow node may need a larger number of bytes than the pointer to the original node. If the memory address of the shadow copy is far from the memory address of the first child node, the delta encoding may require more bytes. If so, the atomic 8 bytes write instruction can not be used to replace the previous pointer. In such a case, instead, all the pointers in the parent node must be encoded again and the entire cache line needs to be updated.

5.5 Journaling

Alternatively, we can employ write-ahead logging (WAL) or journaling (undo logging) techniques. Since journaling incurs less overhead than write-ahead logging for node split operations in clfB-tree, we discuss journaling in this section. Journaling is similar to FAIR algorithm in the sense that both perform in-place update. However, journaling creates a back-up copy and makes changes to the original tree node. The in-place update does not incur an unnecessary upward cascading shadowing problem unless a leaf node overflows.

However, the journaling requires three cache line flushes in clfB-tree, as shown in Figure 25b. (1) First, when a new entry is to be inserted, a copy of the original tree node is saved as a journal. This copy requires a cache line flush. (2)(3) In the next step, we perform in-place update to add the new entry, which also requires another cache line flush. If the system crashes while we are updating the original node, the node can be recovered from the saved journal. (4) After successfully updating the original node, we can delete or truncate the journal, which requires a third cache line flush.

Since the size of clfB-tree is a single cache line, the journaling overhead is not as significant as that of other B-tree variants. That is, if a tree node is 1 KBytes, it requires a maximum of 64 cache line flushes just for step (2). Although previously mentioned, we would like to emphasize that our failure-atomic shift insertion and FAIR node split/merge algorithm require only one and three cache line flushes, respectively.

5.6 Re-Encoding and Journaling

Although FAST and FAIR guarantee failure atomicity without journaling or CoW, they can be used only when the number of bytes used for encoding does not change. That is, if a new entry, for example a sibling pointer, requires more bytes, all entries in the tree node need to be encoded again. We can avoid this overhead if we encode each key and each pointer using a different number of bytes and store the encoding information in a separate array. However, the size of this separate array requires more bytes in the cache line, which results in a decrease of the

degree of tree node. Besides this, we found that such independent encoding/decoding requires bitwise operation, and its overhead is not ignorable and a lot more significant than slot-array overhead in wB-trees [10]. Therefore, whenever clfB-tree node changes its encoding, it creates a journal and we perform in-place update, which calls three cache line flushes, as described above.

5.7 Experiments

For evaluation, we implemented two versions of clfB-tree. The first one is clfB-tree with FAST and FAIR (denoted as clfB-tree(FAIR) that calls one cache line flush for a tree node update and three cache line flushes for a node split. The other clfB-tree, denoted as clfB-tree(RTM), eliminates the necessity of FAIR via copy-on-write and hardware transactional memory. That is, if a node splits, clfB-tree(RTM) creates two copies of split nodes and atomically insert the two nodes into the parent node. If the parent node also splits, we again split the parent using CoW and insert the two pointers to the grand parent node. Therefore, RTM can effectively avoid duplicate writes. We also implemented two versions of wB+-tree, one with slot-only nodes (denoted as wB-tree(slot-only)) and the other with slot+bitmap nodes (denoted as wB-tree(slot+bitmap)). We compiled all implementations using g++ 4.8.5 with an option -O3.

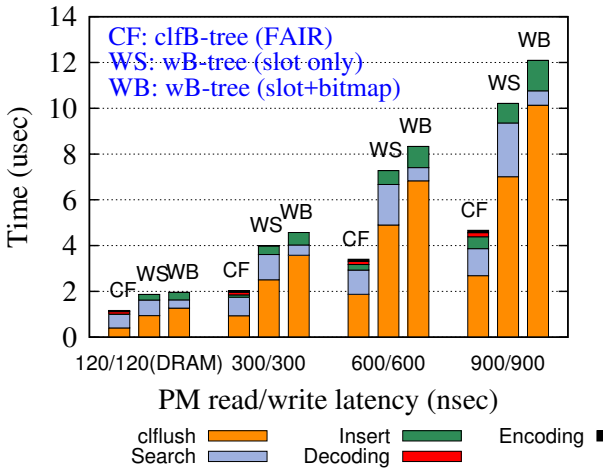
We conduct experiments on a workstation that has four Intel Xeon Haswell-Ex E7-8860 v3 processor (2.20Ghz, 16x32KB instruction cache, 16x32KB data cache, 16x256KB L2 cache, and 40MB L3 cache) and 256GB of DDR3 DRAM. The processors that we use support TSX (Transactional Synchronization Extensions) including Intel' RTM feature. We set CPU frequency scaling governor to performance mode to reduce the standard derivation of experiments.

In order to emulate persistent memory latency, we use software-based persistent memory latency emulator - *Quartz* [57], which models application-perceived PM latency by inserting stall cycles in each predefined time interval, called epoch. The minimum and maximum epochs are set to 5 nsec and 10 nsec in our experiments. Since *Quartz* does not implement write latency emulation [58] in its public distribution, we emulate the write latency by adding `nop` instructions after each `clflush` and `pcommit` instructions as presented in [2, 26, 39]. We do not add write latency for normal `store` instructions because the latency can be hidden by CPU caches.

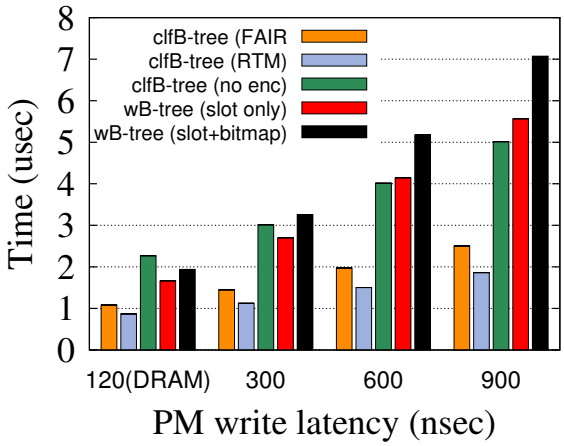
Persistent Memory Latency Effect

In the first set of experiments shown in Figure 26, we insert 1.6 Gbytes of 100 million random key value pairs and measure the average time spent per query while varying read and write latency of persistent memory. We adjust the read latency of persistent memory from 300 nsec, which is the lowest persistent memory latency that we can configure using Quartz in our testbed machine. The write latency of persistent memory is expected to be much higher than DRAM latency. For example, phase change memory is about 2∼5 times slower than DRAM as it has access latencies in the hundreds of nanoseconds [27].

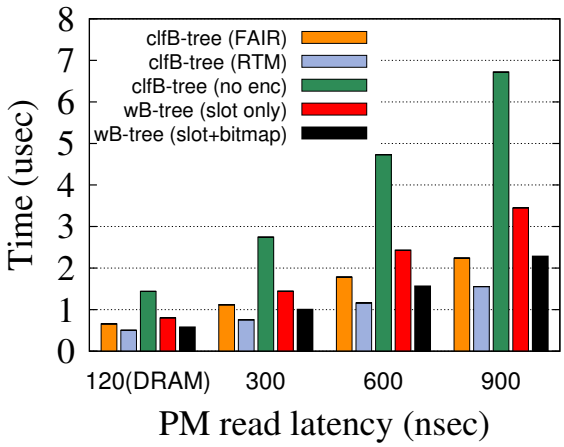
Overall, clfB-tree(FAIR) outperforms wB-trees by a large margin. As we increase the latency,



(a) Insert : Varying Read/Write Latency



(b) Insert : Varying Write Latency



(c) Search : Varying Read Latency

Figure 26: *Performance with Varying Latency of PM (AVG of 5 Runs)*

the performance gap widens because clfB-tree calls less number of cache line flush instructions. The cache line flush accounts for dominant fraction of insertion time for all persistent indexes. When the emulated read and write latency are set to 300 nsec, cache line flush accounts for about 46% of the insertion time of the clfB-tree(FAIR), while it takes about 78% of the insertion time in the wB-tree(slot+bitmap). The overhead of delta encoding in clfB-tree is only less than 3%, and the overhead of delta decoding is also no higher than 6%. [8] For each insertion, we need to search a couple of internal tree nodes to find a leaf node where to insert. The overhead of search in clfB-tree is about 50% (0.93 usec) whereas that of wB-tree(slot-only) is about 40% (1.1 usec) when read latency is 300 nsec. The search overhead increases as we increase the read latency.

In the experiments shown in Figure 26b, we run the same batch insertion workload with more variants of clfB-tree while increasing only the write latency. clfB-tree(RTM) shows the fastest insertion performance. clfB-tree(RTM) is faster than clfB-tree(FAIR) because it does not store a sibling pointer in tree nodes. Having one more encoded pointer requires more frequent re-encoding because if a sibling pointer needs more bytes than the number of bytes currently being used in its tree node. Also, the sibling pointer decreases the number of children. We also measured the performance of clfB-tree that does not use the delta encoding, which is denoted as clfB-tree(no enc). Compared to the other two clfB-trees, clfB-tree(no enc) shows much slower insertion performance because of low degree of tree nodes. Both wB-trees are up to 4.3x slower than clfB-tree(FAIR) when the latency is 1800 nsec. If persistent memory is as fast as DRAM, clfB-tree(FAIR) is 1.53x faster than wB-tree(slot-only) and 1.79x faster than wB-tree(slot+bitmap). wB-tree(slot-only) is faster than wB-tree(slot+bitmap) because its node size is smaller (160 bytes vs. 1 Kbytes), which helps reduce the logging overhead for node splits.

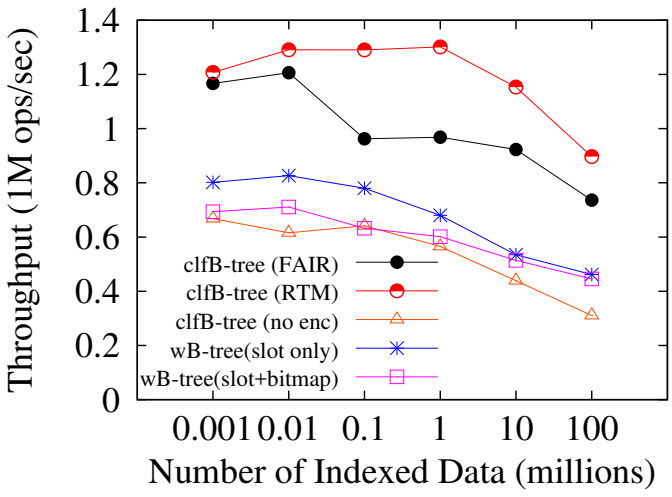
In the experiments shown in Figure 26c, we measure the average search time using the index we created with 100 million key value pairs. Since the search performance is not affected by write latency, we increase only read latency using Quartz. Overall, clfB-tree(RTM) shows the fastest search performance (0.5∼2.73 usec), and clfB-tree(FAIR) and wB-tree(slot+bitmap) show comparable performance (0.65∼3.9 usec and 0.6∼4.2 usec). Although clfB-tree(no enc) has no overhead of encoding and decoding, it suffers from its low degree of nodes.

Overall, clfB-tree consistently outperforms the wB-trees in terms of both insertion and search performance by a large margin in various persistent memory latency configurations.

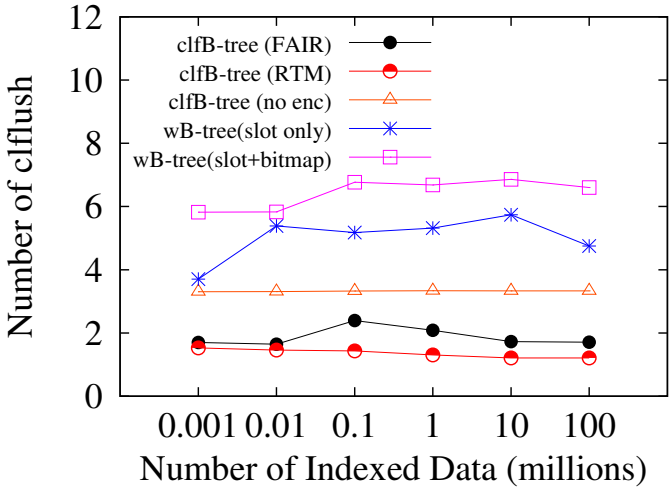
Throughput and Cache Line Flush Overhead

Now, we evaluate the throughput of clfB-tree with varying the number of indexed data. Due to the small degree of tree nodes, the height of clfB-tree is similar to that of wB-tree(slot-only) but much taller than legacy B-trees and wB-tree(slot+bitmap). Compared to wB-tree(slot+bitmap), clfB-tree(RTM) splits tree nodes 3.9 times more frequently when we insert 100 million key value pairs. However, Figure 27b shows the number of indexed data does not significantly affect the

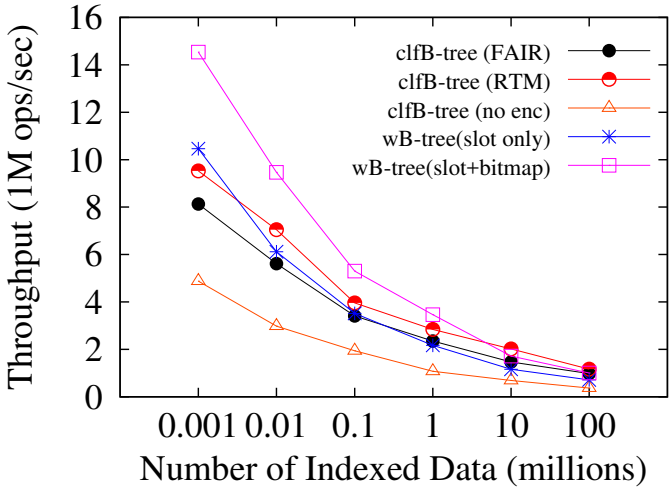
[8] Due to the epoch size limitation of Quartz, the emulated latency could have been injected into a different category.



(a) Insertion Throughput



(b) Number of clflush



(c) Search Throughput

Figure 27: *Performance with Varying Number of Indexed Data (AVG of 5 Runs)*

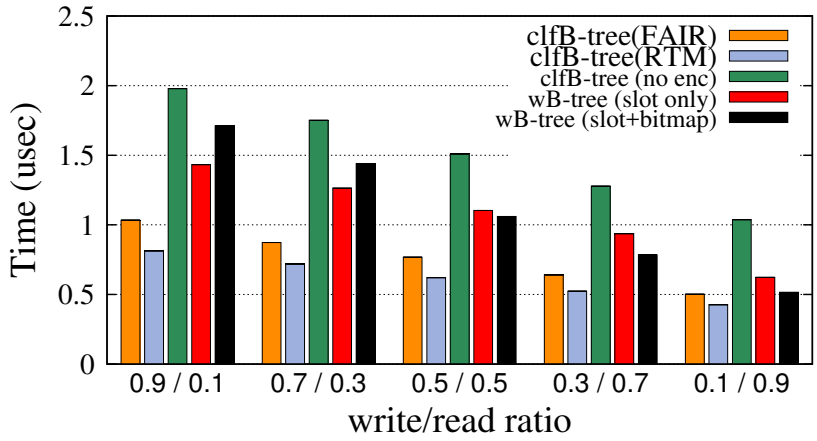


Figure 28: *Mixed workload*

average number of cache line flushes per insertion. This is because 91% of insertions modify only a single leaf node in clfB-tree and the FAST and FAIR algorithms effectively reduce the number of cache line flushes. An insertion into a leaf node in wB-tree(slot-only) and wB-tree(slot+bitmap) call two and four cache line flush instructions respectively [10]. However, if a leaf node splits, wB-tree requires a large number of cache line flush instructions for journaling, creation of a sibling leaf node, and the in-place update, which significantly increase the number of calls to cache line flush instruction. Note that creating a copy of 1K tree node in wB-tree(slot+bitmap) requires at least 32 cache line flushes, and journaling/logging doubles/triples the number.

The clfB-tree(RTM) calls two cache line flush instructions for two CoW leaf nodes, but it does not need logging or journaling as the parent node can update two child pointers atomically. Therefore, the average cache line flushes in clfB-tree(RTM) is about 1.2∼1.5. The clfB-tree(FAIR) requires slightly more cache line flushes (1.7∼2.4). This is not because it requires more cache line flushes per earch split, but because the additional sibling pointer reduces the node utilization and more node split occurs in clfB-tree(FAIR).

Figure 27a shows that the insertion throughput decreases as we insert more data not because of more cache line flushes but because of the taller tree height. The insertion throughput of clfB-tree(RTM) is 1.5x∼1.9x higher than that of wB-tree(slot+bitmap). clfB-tree(FAIR) also shows 1.45x∼1.59x higher throughput than wB-tree(slot+bitmap).

As for the search performance, clfB-trees show comparable performance against wB-trees when the index has more than 10 million data, as shown in Figure 27c. However, when the index size is small, wB-tree(slot+bitmap) outperforms clfB-trees. When we insert just 1,000 records, wB-tree(slot+bitmap) is 1.4x faster than clfB-tree(RTM). This is because such a small number of records need only a few tree nodes in wB-tree. Since wB-tree(slot+bitmap) nodes are large, the index keys are stored in consecutive memory space, which appears to utilize CPU cache better. The node size of wB-tree(slot-only) is also small, hence it shows comparable performance with clfB-trees.

Mixed Workload

While the FAST and FAIR algorithms of clfB-tree is effective for insertion, but the delta encoding is not favorable for search queries. In the experiments shown in Figure 28, we examine the effectiveness of clfB-tree against wB-trees with varying the search and write ratio. Note that, the search and write ratio depends on workloads, and it has been reported that the workloads in SQLite apps in smartphones are write dominant [7, 8, 59]. For the experiments, we first insert 2 million key value pairs, and submit 2 million insert or search queries while varying the ratio. Since insertion is about twice more expensive than search, clfB-tree(RTM) outperforms wB-trees(slot+bitmap) even for search intensive workloads (search 90%, insertion 10%).

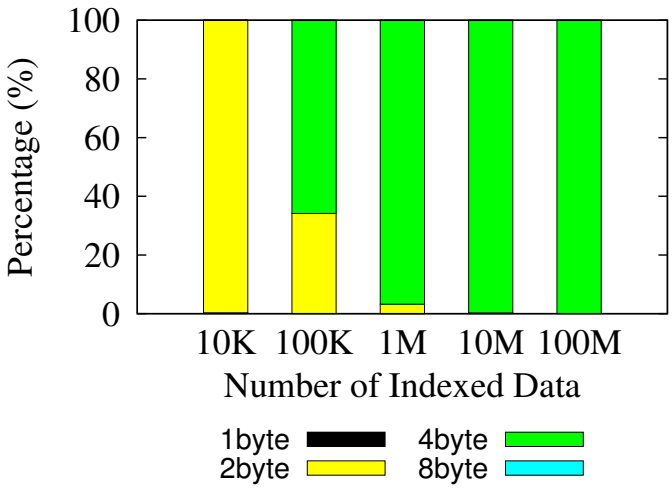
Memory Consumption

In the experiments shown in Figure 29, we show how many bytes are required for encoding each key and child pointer as we increase the number of inserted data. As we insert more data, tree size becomes larger and we need more memory blocks. Therefore, the persistent memory addresses become distant from each other, and the delta encoding needs more bytes for the pointers. However, the delta encoding provides high compression rate, as shown in Figure 29a. When we insert 100 million key-value pairs, 100% of the encoding uses 4 bytes. When we insert 100 thousand key-value pairs, the pointer encoding uses at most 4 bytes; 99.8% of the encoding uses 4 bytes and 0.02% use less than 4 bytes.

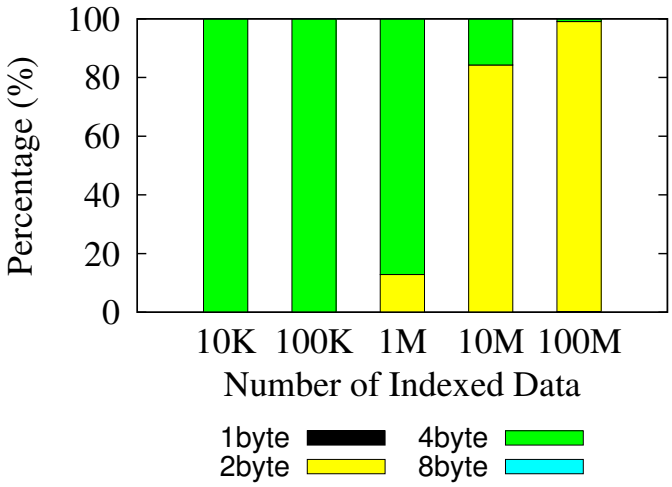
Figure 29b shows the opposite results. When the number of inserted data is small, the keys in a clfB-tree node are often sparse and the key encoding uses more bytes. As we insert more data, the keys, especially in leaf nodes, become denser and use less bytes for the encoding. When we index 1000 key value pairs, most of the pointer encoding uses 2 byte, but most of the key encoding uses 4 bytes. On the contrary, when we index 10 million key value pairs, most of the pointer encoding uses 4 bytes, but most of the key encoding uses 2 bytes.

It should be noted that the key compression rate is dependent on key distribution. In the experiments shown Figure 29c, we vary the number of inserted data and the standard deviation of the key distribution. As the standard deviation is high and the keys are sparse, the compression rate decreases and the node utilization drops. However, overall, we observe the node utilization of clfB-tree (number of encoded entries/number of available slots without encoding) is approximately between 200% and 500%. It should be noted that B-tree node is not often fully utilized. I.e., it is well known that the average node utilization of classic B-tree is 66%. Compared to that number, the differential encoding helps improve the node utilization by 3∼7 times.

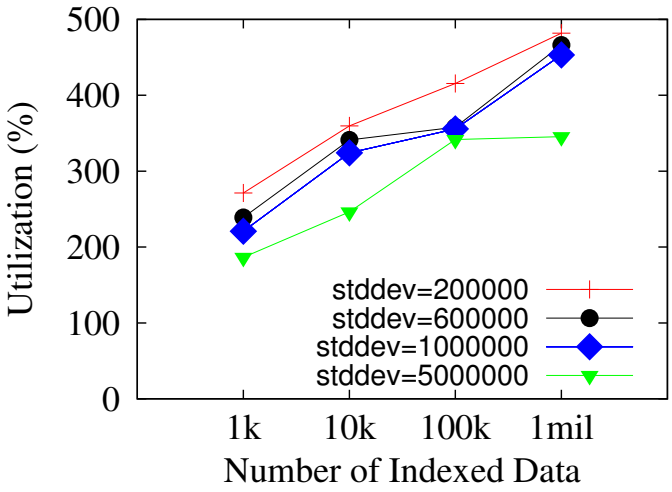
Persistent memory, in comparison to DRAM, is not likely to be cheap. Hence, we show how much clfB-tree can save the memory consumption via delta encoding. Figure 9 shows the number of MBytes used to store the indexing structures. When we insert 100 million 8 byte keys and 8 byte values, wB-tree(slot+bitmap) uses more than 2.6 GBytes of persistent memory



(a) Memory Allocation Locality with Varying the Number of Indexed Data



(b) Indexed Key Locality with Varying the Number of Indexed Data



(c) Node Utilization with Varying the Number of Indexed Data

Figure 29: *Node Utilization Improvement with Differential Encoding*

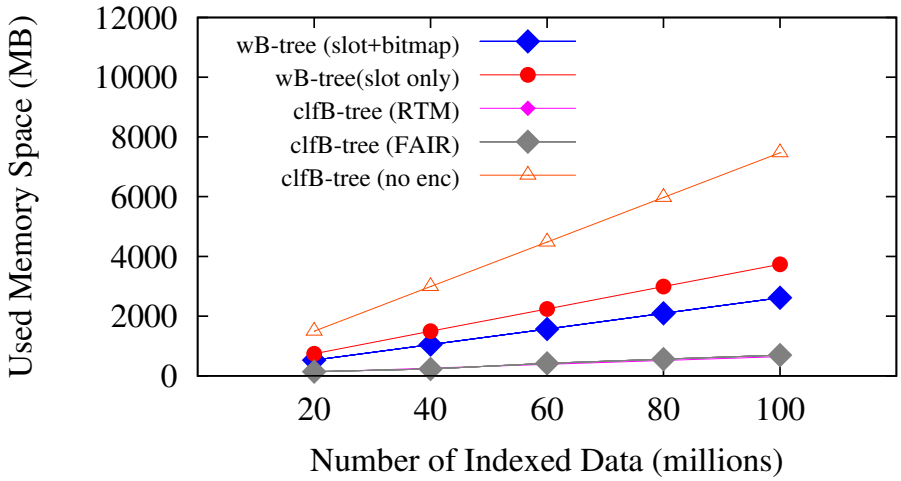


Figure 30: *Memory consumption*

while clfB-tree uses only 0.65 GBytes of persistent memory. That is, clfB-tree can save 75% of memory space compared to wB-tree(slot+bitmap).

5.8 Summary

In this work, we develop a novel persistent B-tree index - clfB-tree (Cache Line Friendly B-tree). To minimize the number of cache line flushes, we set the size of a tree node to a single cache line. Leveraging cache line-fit node size, we develop Failure-Atomic ShifT (FAST) node update algorithm that can sort keys in a tree node with a single cache line flush instruction. We also develop a novel logless but Failure-Atomic In-place tree structure Rebalancing algorithm(FAIR). FAIR helps eliminate duplicate writes caused by logging or journaling and reduce the number of cache line flushes.

Our experimental results show that clfB-tree significantly reduces the number of cache line flushes and outperforms wB-trees in terms of insertion performance while showing comparable search performance. When PM is as fast as DRAM, clfB-tree using FAST and FAIR algorithms shows up to 1.79x faster insertion performance than wB-tree.

VI Persistent Memory Management

Recently, various persistent heap managers and persistent file systems have been proposed for persistent memory management [14, 15, 17, 26–28, 31, 36, 48]. In this work, we develop a memory allocator for persistent memory. The main challenges for persistent memory allocator is persistent problem such as persistent fragmentation, persistent memory leak. To prevent persistent memory leak, previous studies divides the allocation phases into two parts reserve and publish [15,17]. Our work also employ this approach to prevent the persistent memory leak. To prevent the persistent memory fragmentation, segregated list based approach [14, 15] and Log structured approach [31] are used in memory allocation.

6.1 Persistent Memory Development Kit

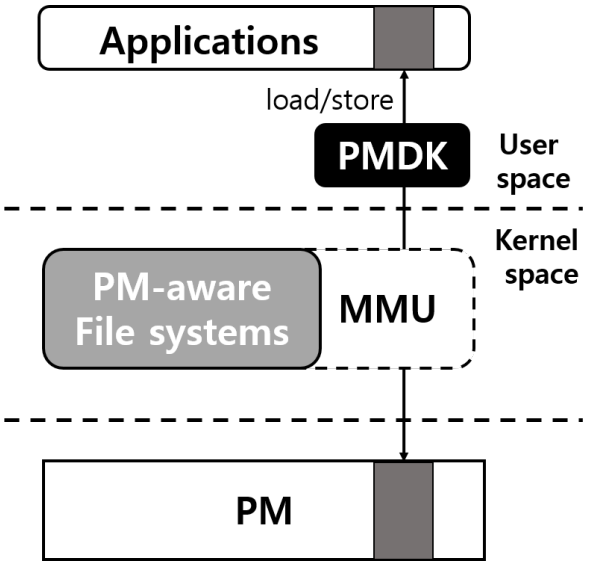


Figure 31: Overview of Persistent Memory Development Kit

Persistent Memory Development Kit(PDMK, formerly known as Non-Volatile Memory Library) is a library for managing Intel DC persistent memory. PMDK is now actively developing, and it is open sourced [15]. There are various packages for persistent memory, the most related library to ours is libpmemobj which is for managing object in persistent memory. In libpmemobj, the allocated spaces are managed as objects called memblock.

As libpmemobj is designed for exploiting byte-addressability of persistent memory, libpmemobj includes memory allocation mechanism and supports transactional functions. In the libpmemobj implementation, the memory allocation process consists of two functions one is palloc_reservation_create(), and the other is palloc_exec_actions() to prevent a persistent memory leak. Once a reservation is done, the allocated persistent region is published to an application. To minimize persistent memory fragmentation, many allocators include PMDK employed segregated lists which divide large chunks to allocation class sizes. Since the similarly

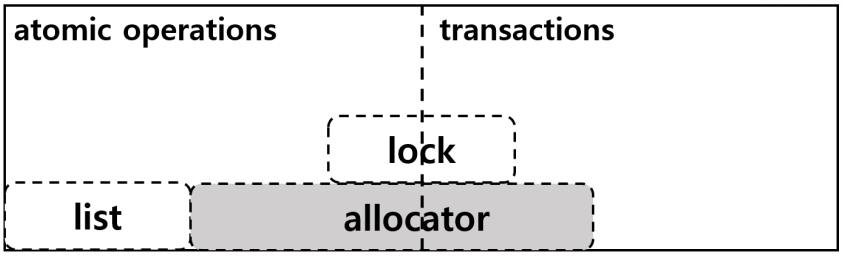


Figure 32: Overview of libpmemobj

sized regions are in the same lists, we can find bestfit sized space more efficiently. In this work, we focus on the data structure for memory allocation in libpmemobj.

6.2 Heap-like memory allocation

Performance Analysis study of libpmemobj

Since libpmemobj's allocator is failure-atomic persistent memory allocator, the metadata of memory allocations should be persistent. In libpmemobj, the basic unit for managing memory space is a chunk. The size of a chunk is 256KB. So if requested memory allocation size is smaller than chunk size, it will be managed by a segregated list with 35 classes of memory allocation. When the memory allocation received a new chunk, the allocation operation will get the size class which is determined by requested allocation size. The chunk will be divided into multiple memory blocks which size is eight times of the size class.

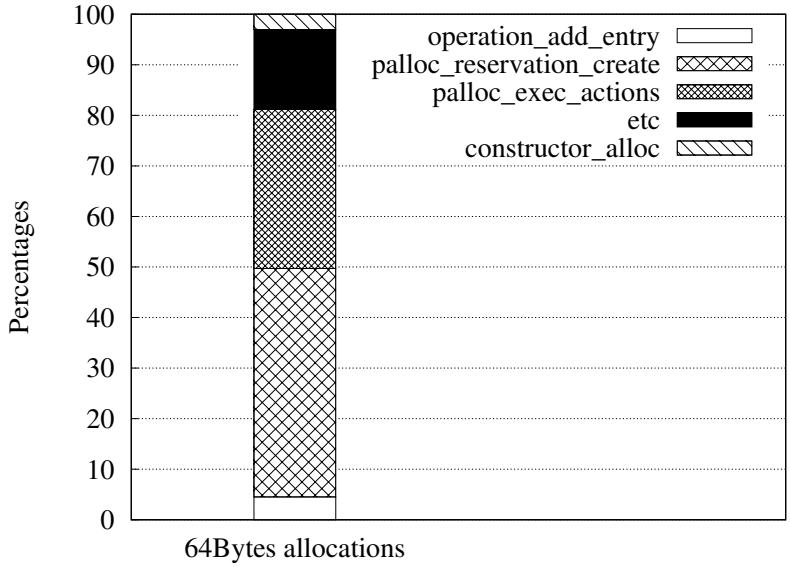


Figure 33: Performance breakdown POBJ_ALLOC with 1Millions 64Bytes allocations

To analyze the cause of performance degradation, we allocate 1 Million of 64bytes objects, and while measuring elapsed time for each function, that is included in allocation functions. Figure 33 shows the performance breakdown of POBJ_ALLOC() function that al-

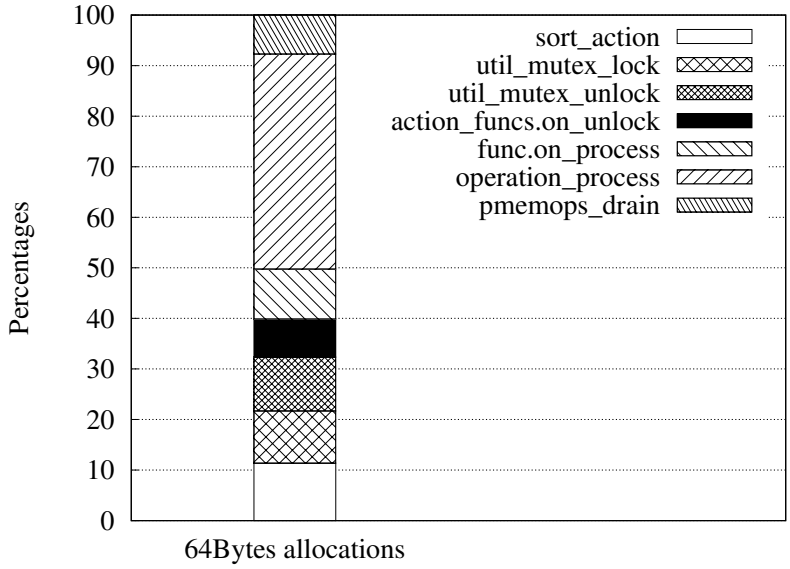


Figure 34: Performance breakdown of palloc_exec_actions with 1Millions 64Bytes allocations

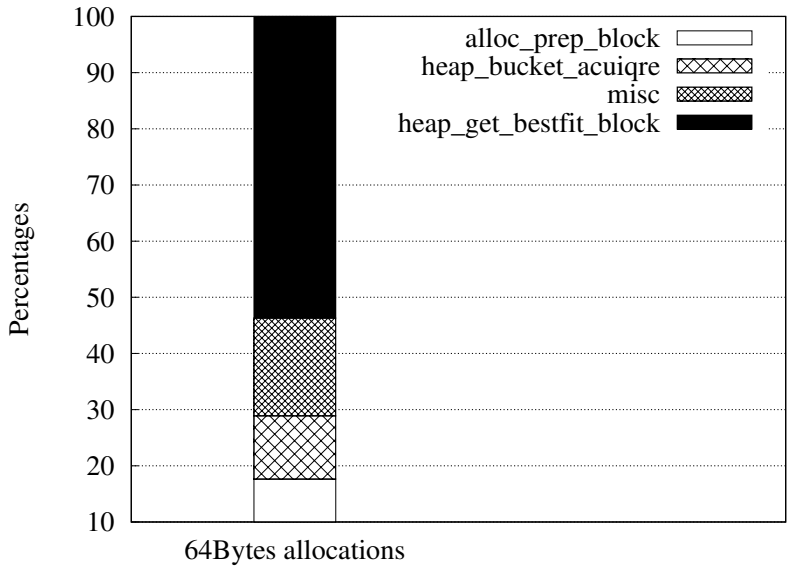


Figure 35: Performance breakdown of palloc_reservation_create with 1Millions 64Bytes allocations

locates the persistent region to the application. As shown Figure 33, POBJ_ALLOC consist of two major functions, palloc_reservation_create() and palloc_exec_actions(). The palloc_reservation_create() function accounts about 45% of total time consumption and execute() function consumes about 31% of total allocation time consumption. We also conduct a performance breakdown experiment of these two functions to point out the main cause of performance degradation.

Figure 34 presents performance breakdown of palloc_exec_actions() functions. This functions includes cache line flush instructions, memory barrier and lock/unlock functions for ex-

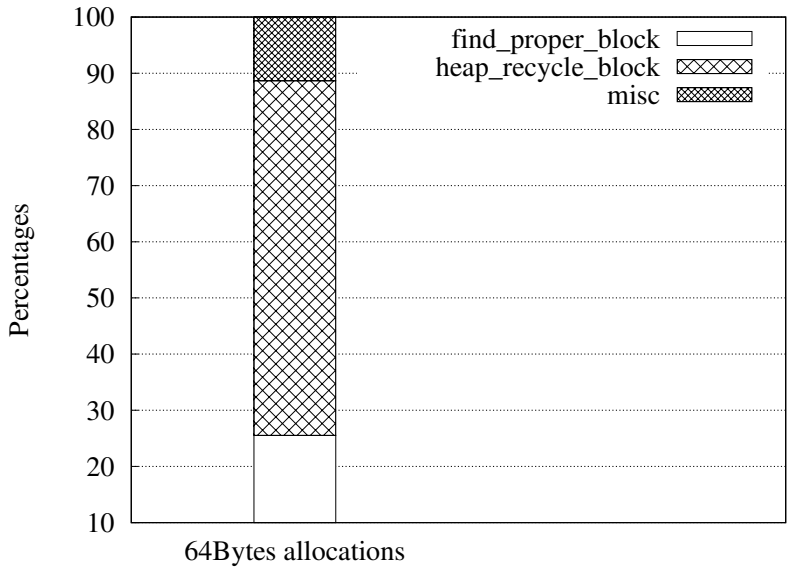


Figure 36: Performance breakdown of segregated fit algorithm with 1Millions of 64Bytes allocations

ecuting the memory operations. The most time consumed function in palloc_exec_actions() function is operation_process() functions. The operation_process() function elapsed of 40% of palloc_exec_actions() function time. This function includes cache line flushes for chunk's metadata. When a memory allocation request asks to allocate persistent memory region, libpmemobj finds bestfit memory space from segregated lists. To prevent memory corruption, the libpmemobj checks, and sets valid bit in the metadata of a chunk that includes the allocated persistent memory space. To persistently store the valid bit status, it calls cache line flush instructions.

Figure 35 shows that time consumption of each functions in the palloc_reservation_create() function. In the palloc_reservation_create() function, the most time consumed function is to get the bestfit blocks quite related to the segregated list. This function makes and manages the segregated list. The heap_bucket_acquire() function includes lock operation to control the concurrent access. In the alloc_prep_block() function, the metadata for allocated space is constructed and persisted. This metadata includes the size of the space and is written by the non-temporal store instruction. The alloc_prep_block() function elapsed 17% of palloc_reservation_create() function time.

In Figure 36, heap_recycle_block() function elapsed 63% of time for heap_get_bestfit_block() function. Since libpmemobj divides a chunk into eight times of size class if the requested size is smaller than eight times of size class space will be wasted. To reuse this the heap_recycle_block() function makes new memory blocks by reusing remaining memory space that is from the allocated memory space.

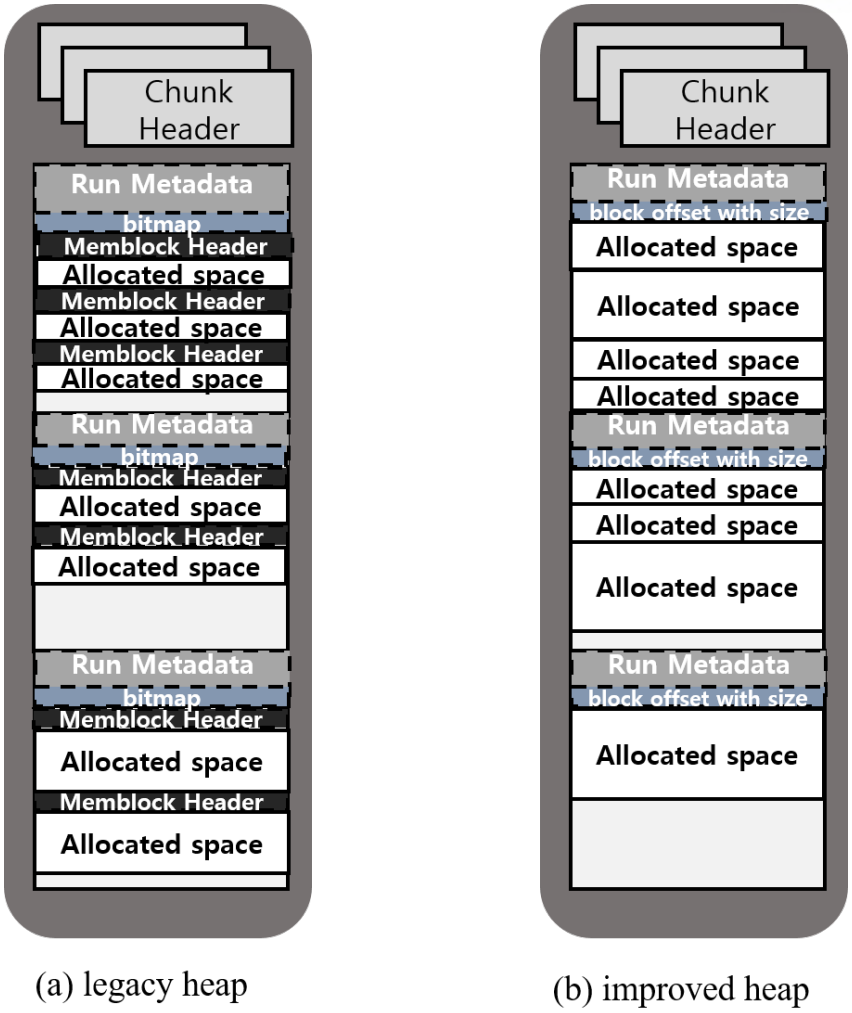


Figure 37: Memory heap layout of legacy PMDK and improved PMDK

Heap-like memory allocation

In our performance analyze study, we find out that the segregated list is performance bottleneck of the libpmemobj. To mitigate this overhead, we developed heap-like memory allocator. The most related previous study to ours is LSNVMM [31]. LSNVMM employed log structured main memory with skiplist that reside on DRAM. The LSNVMM also used thread local cache to avoid the skiplist overhead. Despite skiplist support lock-free read operation, its performance is not that good because of its poor cache locality [13]. In our approach, we replace segregated list based bestfit algorithm to append-only based approach. Also, we remove each memblock's header and store the data of memblock in the memblock's header. The memblock's header only contains the size of objects and extra region. Despite the size and extra region only use 16 bytes, it uses 64Bytes of memory space to prevent false sharing. After store the memblock's header in persistent memory, it updates bitmaps in running metadata of chunk. As the bitmaps show the status of a memblock, it should be persistently stored when the memblock is ready. In our implementation, we only use one cache line flush instructions for block offset+size. Note that,

the default chunk size is 256KB and the allocated objects are aligned in 64bytes.

As the objects are aligned in 64bytes, we need only 2bytes(256KB/64B = 4*1024) to store block offset in a chunk. Also, these structure is for blocks which are smaller than 256KB. So our approach can eliminate the non-temporal store instructions for memblock's header. For huge memory allocation, (bigger than 256KB) we use bestfit algorithm with an additional index as in legacy PMDK. The PMDK employ RAVL [60] that can delete operation without rebalancing in a binary tree. However, as the RAVL does not consider persistent memory, it will need to reconstruct when a system crashes. The current implementation of ours also use RAVL as in PMDK, but we also replace the index to our byte-addressable persistent index as presented in Section V.

Performance Evaluation

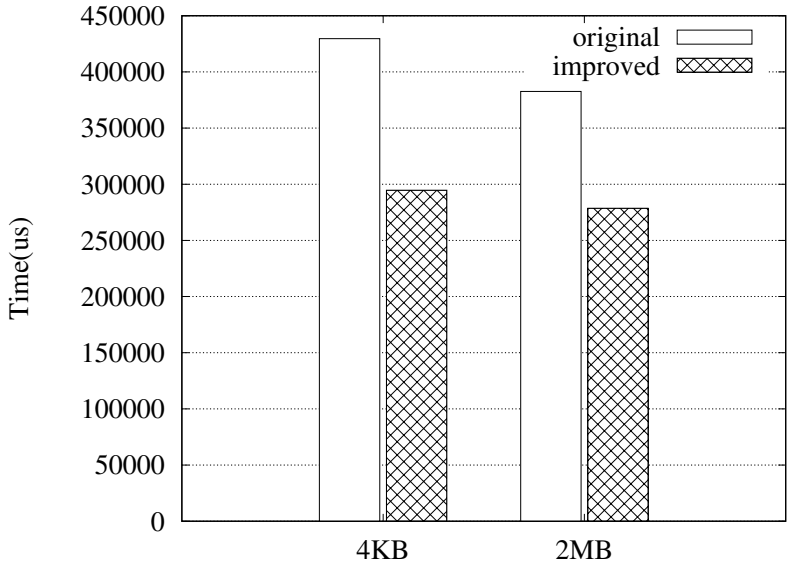


Figure 38: Time Comparison of legacy PMDK and improved PMDK

Figure 38 shows that elapsed time for 1 Million of 64Bytes objects allocations in 4KB based paging systems. Since original PMDK calls cache line flush instructions for bitmaps to validate a newly created object and exploits non-temporal store instructions to directly write the object's header. Ours improved PMDK does not use bitmaps to validate allocated objects. Instead, the improved version uses block offset array to indicate where objects are stored. To persistently store the block offsets, we also use non-temporal store operation. We also conduct the same experiment in 2MB huge page based paging systems. To use huge page we simply use hugetlbfs and memory mapped the memory region. In the 4KB system, we achieve 31% performance improvement against original PMDK. In the 2MB system, the improved version reduces 27% of time consumption for 1Million of 64Bytes of objects.

VII Conclusion

In this dissertation, we investigate how to exploit the non-volatile memory in Transaction Processing System. Since traditional Transaction Processing Systems are well optimized for legacy storage device, it cannot fully exploit the performance of non-volatile memory. As non-volatile memory technologies are now coming as main stream of storage devices, we redesign Transaction Processing Systems for non-volatile memory.

The main cause of the excessive I/O traffics is misaligned interaction between the Transaction Processing Systems such as SQLite and EXT4 in Android Systems. Our LS-MVBT eliminates temporary files for recovery by employing Multi-Version B+-tree and minimize the write traffics in a fsync() calls with five optimizations. The LS-MVBT achieves 70% performance improvement against WAL mode, the fastest original SQLite recovery mode. Furthermore, we present NVWAL that exploits persistent memory to minimize the storage overhead. Our proposed byte-addressable logging, user-level heap manager, and transaction-aware heap management achieve 37% higher throughput than original write-ahead logging on persistent memory. We also show the failure-atomic slotted paging for persistent main memory. The failure atomic slotted paging almost eliminate redundant I/O for guaranteeing data consistency.

The index is also an important part of Transaction Processing System. However, as the traditional index is optimized for block storage device or volatile main memory, they are not well suited for persistent memory. To optimize the index performance, several approaches minimize cache line flushes [9, 10, 14, 16, 22]. However, even the memory I/O granularity is equal to cache line size, they are aligned to the cache line size. It makes additional I/O traffics. To minimize the I/O traffics we developed cache line friendly B+-tree(clf-B+-tree) that aligns node size to cache line size. The clf-B+-tree makes atomic node update with hardware transactional memory possible. Also, it can leverage failure-atomic shift operation and failure-atomic in-place rebalancing techniques to eliminate the additional copy overhead.

In our memory management scheme, we use memory heap structure to minimize the segregated list overhead which is well studied for minimize memory fragmentation [14, 15, 17]. Our performance evaluation shows that our scheme gains up to 31% of performance improvement against original PMDK.

References

[1] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts*. McGraw-Hill, 2005.

[2] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "NVWAL: Exploiting NVRAM in write-ahead logging," in *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[3] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[4] "Sqlite," *http://www.sqlite.org/*.

[5] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, 2014.

[6] K. Lee and Y. Won, "Smart layers and dumb result: Io characterization of an android-based smartphone," in *Proceedings of the 12th International Conference on Embedded Software (EMSOFT 2012)*, 2012.

[7] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *Proceedings of the USENIX Annual Technical Conference*, 2013.

[8] W.-H. Kim, B. Nam, D. Park, and Y. Won, "Resolving journling of journal anomaly in Android I/O: Multi-version B-tree with lazy split," in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2014.

[9] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

[10] S. Chen and Q. Jin, "Persistent B+-Trees in non-volatile main memory," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 7, pp. 786–797, 2015.

[11] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Proceedings of 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.

[12] W.-H. Kim, J. Seo, J. Kim, , and B. Nam, "clfB-tree: Cacheline friendly persistent B-tree for NVRAM," *ACM Transactions on Storage (TOS), Special Issue on NVM and Storage*, 2018.

[13] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *16th USENIX Conference on File and Storage Technologies*, 2018, p. 187.

[14] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, "Memory management techniques for large-scale persistent-main-memory systems," in *Proceedings of VLDB Endowment (PVLDB)*, vol. 10, no. 11. VLDB Endowment, 2017, pp. 1166–1177.

[15] "Persistent memory development kit," *http://pmdk.io/*.

[16] J. Yang, Q. Wei, C. Chen, C. Wang, and K. L. Yong, "NV-Tree: reducing consistency const for NVM-based single level systems," in *Proceedings of the 13th USENIX conference on File and Storage Technologies (FAST)*, 2015.

[17] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory allocation for nvram." in *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015.*, 2015, pp. 61–72.

[18] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won, "WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly," in *Proceedings of the 2015 USENIX Anual Technical Conference*, 2015.

[19] H. Luo, L. Tian, and H. Jiang, "qNVRAM: quasi non-volatile ram for low overhead persistency enforcement in smartphones," in *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[20] G. Oh, S. Kim, S.-W. Lee, and B. Moon, "Sqlite optimization with phase change memory for mobile applications," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1454–1465, 2015.

[21] J.-H. Park, G. Oh, and S.-W. Lee, "Sql statement logging for making sqlite truly lite," in *Proceedings of VLDB Endowment (PVLDB)*, vol. 11, no. 4. VLDB Endowment, 2017, pp. 513–525.

[22] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write optimal radix tree for persistent memory storage systems," in *Proceedings of the 15th USENIX conference on File and Storage Technologies (FAST)*, 2017.

[23] P. Zuo and Y. Hua, "A write-friendly hashing scheme for non-volatile memory systems," in *Proceedings of the 33st International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[24] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 461–476. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/zuo

[25] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[26] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[27] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making persisten objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[28] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, N. Binkert, and P. Ranganathan, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proceedings of the ACM Conference on Timely Results in Operating Systems (TRIOS)*, 2013.

[29] T. Hwang, J. Jung, and Y. Won, "Heapo: Heap-based persistent object store," *ACM Transactions on Storage (TOS)*, vol. 11, no. 1, 2014.

[30] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, Cheng-Yuan, and M. Wang, "NVM duet: unified working memory and persistent store architecture," in *19th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[31] Q. Hu, J. Ren, A. Badam, and T. Moscibroda, "Log-structured non-volatile main memory," in *Proceedings of the USENIX Annual Technical Conference*, 2017.

[32] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 677–694.

[33] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion B-tree," *VLDB Journal*, vol. 5, no. 4, pp. 264–275, Dec. 1996.

[34] B. Sowell, W. Golab, and M. A. Shah, "Minuet: A scalable distributed multiversion b-tree," in *Proceedings of the VLDB Endowment, Vol. 5, No. 9*, 2012.

[35] G. Wu and X. He, "Reducing ssd read latency via nand flash program and erase suspension," in *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, 2012.

[36] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)*, 2014, pp. 15:1–15:15.

[37] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 265–276.

[38] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Implications of cpu caching on byte-addressable non-volatile memory programming," *http://www.hpl.hp.com/techreports/2012/HPL-2012-236.pdf*, 2012.

[39] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 707–722.

[40] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "Rewind: Recovery write-ahead system for in-memory non-volatile data-structures," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 497–508, 2015.

[41] S. D. Viglas, "Data management in non-volatile memory," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1707–1711.

[42] Z. Wang, H. Qian, J. Li, and H. Chen, "Using restricted transactional memory to build a scalable in-memory database," in *ACM SIGOPS/Eurosys European Conference on Computer Systems (EuroSys)*, 2014.

[43] M. Li and P. P. C. Lee., "Toward i/o-efficient protection against silent data corruptions in raid arrays," in *Proceedings of the 30th International Conference on Massive Storage Systems and Technology (MSST)*, 2014.

[44] S. Agarwal, R. Garg, M. S. Gupta, , and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proceedings of the 18th annual international conference on Supercomputing*, 2004.

[45] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini, "Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers." in *Proceedings of the ACM/IEEE SC2005 Conference*, 2005.

[46] J. Lee, K. Kim, and S. Cha, "Differential logging: A commutative and associative logging scheme for highly parallel main memory database," in *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, 2001.

[47] S.-W. Lee and B. Moon, "Design of flash-based dbms: An in-page logging approach," in *Proceedings of 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007.

[48] X. Wu and A. L. N. Reddy, "SCMFS: A file system for storage class memory," in *Proceedings of the ACM/IEEE SC2011 Conference*, 2011.

[49] T. Lee, D. Kim, H. Park, and S. Yoo, "Fpga-based prototyping systems for emerging memory technologies," in *Proceedings of the 25th IEEE International Symposium on Rapid System Prototyping (RSP)*, 2014.

[50] "OpenNVRAM," *http://opennvram.org/*.

[51] "Mobibench," *https://github.com/ESOS-Lab/Mobibench*.

[52] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009.

[53] A. Ailamaki, D. J. DeWitt, and M. D. Hill, "Data page layouts for relational databases on deep memory hierarchies," *VLDB Journal*, vol. 11, no. 3, pp. 198–215, Nov. 2002.

[54] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. New York, NY, USA: McGraw-Hill, Inc., 2005.

[55] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *Proceedings of the 30th International Conference on Data Engineering (ICDE)*, 2014.

[56] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[57] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *15th Annual Middleware Conference (Middleware '15)*, 2015.

[58] "Quartz," *https://github.com/HewlettPackard/quartz*.

[59] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2013.

[60] S. Sen, R. E. Tarjan, and D. H. K. Kim, "Deletion without rebalancing in binary search trees," *ACM Transactions on Algorithms (TALG)*, vol. 12, no. 4, p. 57, 2016.

Acknowledgements

First of all, I would like to express my gratitude to my advisor Professor Beomseok Nam. His dedicated guidance and generous supports always encouraged me during the Ph.D. study. His guidance always has made me in the right way. This dissertation would not have been possible without his mentorship.

I am very grateful to Professor Sam H. Noh who is my co-advisor. It was very honor to work with him. His knowledge and insight helped my dissertation have more strengthened.

I would like to thank the committee member of my dissertation, Professor Youjip Won, Professor Young-ri Choi, and Professor Woongki Baek. They provided valuable comments and feedback to improve the quality of this dissertation.

I would also like to thank my close lab fellows Jinwoong Kim and Deukyeon Hwang for share their time in my graduate studies. They made my 6 years of graduate studies more funny and valuable. I would also like to thank Hyunsub Song, Hyeonho Song, J.hyun Kim, Sekwon Lee, and Jihye Seo for sharing their insights in system software research for persistent memory. Also, I would like to thank CISSR members.

Finally, I would like to special thank my family for their support and unconditional love.